The Art of Computational Science

The Kali Code

vol. 4

The N-Body Problem:
From Leapfrog to Runge-Kutta

Piet Hut and Jun Makino

September 14, 2007

Contents

Preface

We continue the dialogue in the previous two volumes, where Alice and Bob developed a gravitational 2-body code, containing a wide choice of integrators. In the current volume, they modify their code, to grow it into a true N-body version. They try out various choices of implementation, and test each version using the recently discovered solution for the 3-body problem, where all three particles follow each other in a figure eight. Toward the end of the book, they simulate a cold collapse of eight particles, first without softening, then with softening.

0.1 Acknowledgments

Besides thanking our home institutes, the Institute for Advanced Study in Princeton and the University of Tokyo, we want to convey our special gratitude to the Yukawa Institute of Theoretical Physics in Kyoto, where this volume was written, during a visit in May 2004, made possible by the kind invitations to both of us by Professor Masao Ninomiya.

We thank Michele Trenti and Jason Underdown for their comments on the manuscript.

Piet Hut and Jun Makino

Kyoto, July 2004

Chapter 1

An N-Body Code

1.1 First Attempt

Bob: Hi, Alice! How are things?

Alice: I'm fine. But I can see from your face that you can't wait to show me something. Did you do some more coding?

Bob: Well, yes, some.

Alice: Don't tell me you wrote a whole N-body implementation?

Bob: Well, yes.

Alice: Now that's impressive! Even though you told me it wasn't going to be *that* much work.

Bob: It was more work than I thought, I admit, but not terribly much more work.

Alice: I guess that 'more' is a relative concept. Can you show me what you did?

Bob: My pleasure. Here is the whole listing. I have called it `rknbody1.rb`, `rk` for Runge-Kutta, and `1` because it is my first attempt, and I'm sure we'll come up with some modifications and improvements.

```
require "vector.rb"

class Body

  attr_accessor :mass, :pos, :vel, :nb

  def initialize(mass = 0, pos = Vector[0,0,0], vel = Vector[0,0,0])
```

```
    @mass, @pos, @vel = mass, pos, vel
  end

  def acc
    a = @pos*0                      # null vector of the correct length
    @nb.body.each do |b|
      unless b == self
        r = b.pos - @pos
        r2 = r*r
        r3 = r2*sqrt(r2)
        a += r*(b.mass/r3)
      end
    end
    a
  end

  def ekin                          # kinetic energy
    0.5*(@vel*@vel)
  end

  def epot                          # potential energy
    p = 0
    @nb.body.each do |b|
      unless b == self
        r = b.pos - @pos
        p += -@mass*b.mass/sqrt(r*r)
      end
    end
    p
  end

  def to_s
    "  mass = " + @mass.to_s + "\n" +
    "   pos = " + @pos.join(", ") + "\n" +
    "   vel = " + @vel.join(", ") + "\n"
  end

  def pp               # pretty print
    print to_s
  end

  def simple_print
    printf("%24.16e\n", @mass)
    @pos.each{|x| printf("%24.16e", x)}; print "\n"
    @vel.each{|x| printf("%24.16e", x)}; print "\n"
  end
```

```
  def simple_read
    @mass = gets.to_f
    @pos = gets.split.map{|x| x.to_f}.to_v
    @vel = gets.split.map{|x| x.to_f}.to_v
  end

end

class Nbody

  attr_accessor :time, :body

  def initialize(n=0, time = 0)
    @time = time
    @body = []
    for i in 0...n
      @body[i] = Body.new
      @body[i].nb = self
    end
  end

  def evolve(integration_method, dt, dt_dia, dt_out, dt_end)
    nsteps = 0
    e_init
    write_diagnostics(nsteps)

    t_dia = dt_dia - 0.5*dt
    t_out = dt_out - 0.5*dt
    t_end = dt_end - 0.5*dt

    while @time < t_end
      send(integration_method,dt)
      @time += dt
      nsteps += 1
      if @time >= t_dia
        write_diagnostics(nsteps)
        t_dia += dt_dia
      end
      if @time >= t_out
        simple_print
        t_out += dt_out
      end
    end
  end
```

```
def forward(dt)
  old_acc = []
  @body.each_index{|i| old_acc[i] = @body[i].acc}
  @body.each{|b| b.pos += b.vel*dt}
  @body.each_index{|i| @body[i].vel += old_acc[i]*dt}
end

def leapfrog(dt)
  @body.each{|b| b.vel += b.acc*0.5*dt}
  @body.each{|b| b.pos += b.vel*dt}
  @body.each{|b| b.vel += b.acc*0.5*dt}
end

def rk2(dt)
  old_pos = []
  @body.each_index{|i| old_pos[i] = @body[i].pos}
  half_vel = []
  @body.each_index{|i| half_vel[i] = @body[i].vel + @body[i].acc*0.5*dt}
  @body.each{|b| b.pos += b.vel*0.5*dt}
  @body.each{|b| b.vel += b.acc*dt}
  @body.each_index{|i| @body[i].pos = old_pos[i] + half_vel[i]*dt}
end

def rk4(dt)
  old_pos = []
  @body.each_index{|i| old_pos[i] = @body[i].pos}
  a0 = []
  @body.each_index{|i| a0[i] = @body[i].acc}
  @body.each_index{|i|
    @body[i].pos = old_pos[i] + @body[i].vel*0.5*dt + a0[i]*0.125*dt*dt}
  a1 = []
  @body.each_index{|i| a1[i] = @body[i].acc}
  @body.each_index{|i|
    @body[i].pos = old_pos[i] + @body[i].vel*dt + a1[i]*0.5*dt*dt}
  a2 = []
  @body.each_index{|i| a2[i] = @body[i].acc}
  @body.each_index{|i|
    @body[i].pos = old_pos[i] + @body[i].vel*dt +
                   (a0[i]+a1[i]*2)*(1/6.0)*dt*dt}
  @body.each_index{|i| @body[i].vel += (a0[i]+a1[i]*4+a2[i])*(1/6.0)*dt}
end

def ekin                        # kinetic energy
  e = 0
  @body.each{|b| e += b.ekin}
  e
```

```
  end

  def epot                        # potential energy
    e = 0
    @body.each{|b| e += b.epot}
    e/2                           # pairwise potentials were counted twice
  end

  def e_init                      # initial total energy
    @e0 = ekin + epot
  end

  def write_diagnostics(nsteps)
    etot = ekin + epot
    STDERR.print <<END
at time t = #{sprintf("%g", time)}, after #{nsteps} steps :
  E_kin = #{sprintf("%.3g", ekin)} ,\
 E_pot =  #{sprintf("%.3g", epot)} ,\
 E_tot = #{sprintf("%.3g", etot)}
             E_tot - E_init = #{sprintf("%.3g", etot - @e0)}
  (E_tot - E_init) / E_init = #{sprintf("%.3g", (etot - @e0)/@e0 )}
END
  end

  def pp                                # pretty print
    print "     N = ", @body.size, "\n"
    print "  time = ", @time, "\n"
    @body.each{|b| b.pp}
  end

  def simple_print
    print @body.size, "\n"
    printf("%24.16e\n", @time)
    @body.each{|b| b.simple_print}
  end

  def simple_read
    n = gets.to_i
    @time = gets.to_f
    for i in 0...n
      @body[i] = Body.new
      @body[i].nb = self
      @body[i].simple_read
    end
  end
```

```
end
```

1.2 Driver

Alice: I recognize the overall structure. Can you show me the driver program, to give me an idea where to start?

Bob: Here it is, `rknbody1_driver.rb`

```
require "rknbody1.rb"

include Math

dt = 0.0001          # time step
dt_dia = 10          # diagnostics printing interval
dt_out = 10          # output interval
dt_end = 10          # duration of the integration
##method = "forward"   # integration method
##method = "leapfrog"   # integration method
##method = "rk2"   # integration method
method = "rk4"   # integration method

STDERR.print "dt = ", dt, "\n",
      "dt_dia = ", dt_dia, "\n",
      "dt_out = ", dt_out, "\n",
      "dt_end = ", dt_end, "\n",
      "method = ", method, "\n"

nb = Nbody.new
nb.simple_read
nb.evolve(method, dt, dt_dia, dt_out, dt_end)
```

Alice: The last three lines are almost exactly the same as the last three lines in the driver for our pseudo-one-body integrator:

```
b = Body.new
b.simple_read
b.evolve(method, dt, dt_dia, dt_out, dt_end)
```

apart for three `n`'s and one `N`. The rest is exactly the same, with of course an extra `n` in the file name that is being required on top. A very minimal change.

Bob: Yes, I preferred to stay as close as I could to our initial 2-body relative coordinates integrator.

Alice: I'm very curious to see how you integrated the combined equations of motion for an arbitrary number of particles, but let me follow the logic of the driver first, to feel my way around in the program flow. You start by creating a whole N-body system, and then reading in the data. And besides the old class `Body`, you have introduced a class `Nbody`, to contain the data for that whole system.

Bob: Yes. The initializer has two parameters, as you can see:

```
def initialize(n=0, time = 0)
  @time = time
  @body = []
  for i in 0...n
    @body[i] = Body.new
    @body[i].nb = self
  end
end
```

By default an N-body system is created empty, containing 0 particles, at starting at time 0, but if you like, you can create it with several particles right from the start. In our case, the driver creates an empty default system, leaving it up to the `simple_read` input function to create the necessary particle slots, depending on how many are needed to store the input data.

Alice: Since all particles share the same time step in this code, it makes sense to make the shared simulation time an instance variable `@time` for the `Nbody` class. And `@body` must be an array of `Body` instances, since if you specify `n` particles to be present, you fill the array by creating `n` new bodies, from `@body[0]` through `@body[n-1]`. But what about the `for` loop? Ah, yes, I remember now: in Ruby `0...n` means that you exclude `n`, while `0..n` does indeed include `n`.

Bob: A handy short notation, but like every short notation, potentially confusing if you're not yet used to it.

1.3 Input

Alice: But what about this line

```
@body[i].nb = self
```

Bob: I have given the `Body` class an extra instance variable `@nb`, which for each body instance will contain the address of the parent, an instance of the

Nbody class. In that way, each Body daughter is doubly linked to her Nbody parent. The parent can call the appropriate daughter, by selecting her from the @body[] array, and the daughter can call the parent directly through her own @nb variable. Remember that the expression self gives the address of the Nbody instance itself, which then gets handed down to each body.

Alice: Hmm. In general, I am quite wary of doubly linked list. It is all too easy to change the link in one direction and to forget to change the link in the other direction, or to change it in the wrong way.

Bob: Typical one-off errors that often happen in C and C++ are less likely to occur in Ruby, because so much of the bookkeeping is handled behind the scenes, as long as you don't confuse 0...n and 0..n. But I see your point, and perhaps we should change that, later on. For now, let's just go through the code, and then we can decide whether it will be easy to unlink the backward pointers from daughter to parent.

My motivation to provide backward links was to give each daughter the possibility to communicate with her siblings. If one daughter wants to compute her acceleration, she would need to find the positions of all other daughters, and the simplest way to do that, I thought, was to give her a way to ask her parents how to find all the others.

Alice: Fine for now. In the simple_read program you also provide those backward links for each particle, after which you invoke the simple_read method for that particle:

```
def simple_read
  @mass = gets.to_f
  @pos = gets.split.map{|x| x.to_f}.to_v
  @vel = gets.split.map{|x| x.to_f}.to_v
end
```

The format you have chosen is to start with the number of particles and the time, followed by the data for each particle.

Bob: Yes. It seemed safer to tell the input routine how many particles to expect, rather than to let it read in everything to the end of the file. In some cases you might want to store more than one snapshot, for example, in one file.

In fact, my code normally will output a series of snapshots, one after each dt_out interval, just as we did it for the single pseudo-body case. This will make it possible to restart a run: you can later sequentially read in a number of those snapshots, each with a nb.simple_read statement in the driver, selecting the proper one by checking the time variable specified.

For example, when you invoke the code with:

```
dt_out = 5
```

```
dt_end = 10
nb = Nbody.new
nb.simple_read
nb.evolve(method, dt, dt_dia, dt_out, dt_end)
```

you can continue the run from the output of this first run, by reading in that output and discarding the first snapshot:

```
nb = Nbody.new
nb.simple_read
nb.simple_read
nb.evolve(method, dt, dt_dia, dt_out, dt_end)
```

Alternatively, and more safely, you could check for the time:

```
nb = Nbody.new
nb.simple_read
while (nb.time < 10)
  nb.simple_read
end
```

It would be better to use a slightly smaller value than 10, if you want to pick up the snapshot corresponding to `nb.time = 10`, since it is quite possible that it will have been output at `nb.time = 9.99999` or so. Also, you would want to check whether the snapshot time is not too far beyond `nb.time = 10`. But those are details.

Chapter 2

N-Body Integrators

2.1 Inspecting the Leapfrog

Alice: Yes, I get the idea, and that all makes a lot of sense. And now that we understand how the data get read in, let's see what will happen with them.

In `rknbody1.rb`, I see that you have shifted all the integration methods from the `Body` to the `Nbody` class, as well as the `evolve` function that calls them.

Bob: The `evolve` function orchestrates the whole integration process, and it is called by the driver, which only knows about the one `Nbody` instance that it has created. So it is logical to put the `evolve` method inside the `Nbody` class. And since `evolve` calls the various integration methods, it also seemed logical to have `leapfrog`, `rk2`, and so on, reside there.

Alice: I could imagine an alternative, where each particle is given the freedom to use its own integration method, in which case you would want to shift those methods back into the `Body` class, but that would make more sense when you use an individual time step algorithm, where each particle has its own time step length. For the simple shared time step case that we are starting with, your choice is surely the best.

Bob: I could imagine many things, but coding them takes more time than imagining them! I do like the idea of relatively autonomous particles, integrating themselves as they want, with stars in denser regions having perhaps more specialized integrators, but not today.

Alice: Looking at `evolve`, I see almost exactly the same function that we used for the two-body problem. The only difference is that now the time is an instance variable for the `Nbody` class, which means that we don't have to pass the time as an argument to the `write_diagnostics` method.

Bob: Yes. If I would have left the time as a normal variable that would be passed around, the `evolve` method would have been *exactly* the same. A nice example of recycling code: whether you are dealing with one pseudo particle or with N particles, the top level instructions are basically the same.

Alice: But of course the actual work is different, and in our case more complicated. The forward Euler implementation is a bit hard to recognize, at first sight. Let me start with the new leapfrog method, which looks more familiar. The two-body version was:

```
def leapfrog(dt)
  @vel += acc*0.5*dt
  @pos += @vel*dt
  @vel += acc*0.5*dt
end
```

while now we have

```
def leapfrog(dt)
  @body.each{|b| b.vel += b.acc*0.5*dt}
  @body.each{|b| b.pos += b.vel*dt}
  @body.each{|b| b.vel += b.acc*0.5*dt}
end
```

This is easy to understand: for each body, basically the same actions are taken as was the case for our single pseudo-body, containing the relative position information for the two-body case.

Bob: The difference being that, invisibly at this level, the `Body` method `acc`, which computes the acceleration, has to ask all other particles for their position.

2.2 Acceleration

Alice: Indeed, `acc` has grown quite a bit bigger. In the two-body case, we started with

```
def acc
  r2 = @pos*@pos
  r3 = r2*sqrt(r2)
  @pos*(-@mass/r3)
end
```

and your new N-body version reads:

```
def acc
  a = @pos*0                          # null vector of the correct length
  @nb.body.each do |b|
    unless b == self
      r = b.pos - @pos
      r2 = r*r
      r3 = r2*sqrt(r2)
      a += r*(b.mass/r3)
    end
  end
  a
end
```

Bob: The main difference is the loop that our body has to execute over all other bodies. It is here that I am using my backpointer `@nb` that links back to the parent `Nbody` instance. In that way, the array of bodies becomes visible for our particular body as `@nb.body`, and it is this array over which we iterate using the familiar `each` construct.

Alice: And you are excluding the body itself from the loop, to avoid getting an infinitely large self interaction, through the line:

```
    unless b == self
```

But what exactly are you comparing? I am used to the C notation where `==` compares two numbers. In Ruby too, when both numbers are equal, the statement returns `true`, and if not, it returns `false`. But what are the two numbers being compared here?

Bob: In Ruby, each object, that is each instance of any class, has a unique id number, a machine-defined number that is guaranteed to be different for two different objects. We don't have to know anything about what that number is, or how it is represented. All we need to know is that we can rely on it being different for two different particles.

Alice: But this unique identification number has a different status from that of normal numbers, such as integers or floating point numbers, I presume. If I write `a == b` for two variables, Ruby compares the values of these two variables, not their id numbers. If Ruby would always use the `==` operator to compare object id numbers, then `a == b` would always result in `false`, whenever the two variables would be different, whether they have the same values or not.

Bob: Yes, you are right. I had not thought about that. In the case of numbers, or strings for that matter, the `==` operator must be overloaded so as to override the default behavior, which is comparing id numbers. Interesting! I had just used this expression, since it seemed reasonable, and does the right thing. But now that you ask me, yes, there must be different types of overloading going on for different classes. In other words, many different classes must define their own `==` method.

Alice: The good thing about Ruby is that everything happens so naturally, in such a transparent way. But a consequence is that you often don't appreciate all that is going on behind the scenes. Coming back to the statement above, this line is filtering out particle pair combinations where both particles have the same identity.

Bob: Indeed. particles are not allowed to interact with themselves. For all other particle pairs, we compute the acceleration in a similar way as before. The main difference is that the vector connecting the two bodies is not given, as was the case for the two-body problem, where there was only one relative vector. Here we compute the vector pointing from the calling particle to the called particle first, as follows:

```
r = b.pos - @pos
```

Alice: And the acceleration *seems* to have the same mass dependence in both cases, the two-body and the N-body case, but here appearances deceive: in the two-body case we had an equation of moment for our pseudo particle, while here we are now dealing with real particles.

Bob: Yes, I thought about that carefully. Actually, the tricky thing is to get the two-body case right, where it is easy to make a mistake, as we saw when I was a bit too quick in coding up the diagnostics there. For the N-body case, in contrast, it is all a piece of cake. The line

```
a += r*(b.mass/r3)
```

Directly implements Newton's law of gravity.

2.3 Newtonian Gravity

Alice: When we present this to our students, it would be good to summarize the connection specifically. To wit: the expression for the acceleration felt by particle i is given by summing together the Newtonian gravitational attraction of all other particles j, where both i and j take on values from 1 up to and including N, according to the text books. In our case, of course, we label particles with numbers starting from 0 and running up to and including *N-1*, since that is Ruby's default way of numbering arrays. Let's write the equations accordingly:

$$\frac{d^2}{dt^2}\mathbf{r}_i = G\sum_{\substack{j=0\\ j\neq i}}^{N-1} M_j\frac{\mathbf{r}_j-\mathbf{r}_i}{|\mathbf{r}_j-\mathbf{r}_i|^3} \tag{2.1}$$

Here M_j and $\mathbf{r}_j$ are the mass and position vector of particle j, and G is the gravitational constant.

When I write this equation on a black board in front of a class, there is always someone who asks me where the power of 3 in the denominator comes from, given that Newtonian gravity is an inverse square law, and therefore should be proportional to the power 2 of the distance, in the denominator. To bring out the inverse square nature of gravity, I then write $\mathbf{r}_{ji} = \mathbf{r}_j - \mathbf{r}_i$, with $r_{ji} = |\mathbf{r}_{ji}|$, after which I define the unit vector $\hat{\mathbf{r}}_{ji} = \mathbf{r}_{ji}/r_{ji}$. This allows the above equation to be written as:

$$\mathbf{a}_i = G \sum_{\substack{j=0 \\ j \neq i}}^{N-1} \frac{M_j}{r_{ji}^2} \, \hat{\mathbf{r}}_{ji} \tag{2.2}$$

with the expected power of 2 in the denominator.

Finally, I note that the summation excludes self-interactions: every particle feels the forces of the other $N-1$ particles, but not its own force, which, as we already mentioned, would be infinitely large in case of a point mass.

Bob: That's a nicely crisp summary.

Alice: What is really nice in our Ruby implementation, is that we *never* have to introduce the counters i and j that are so ubiquitous in any N-body code I have ever seen. Just as we could dispense with the k variable for the components of a vector, we can avoid the other two counters by asking arrays and vectors to just loop over themselves.

And this is one of the features that makes Ruby eminently suited for prototyping and development work in general. Whether Ruby will be used eventually for an industrial-strength production-type code, that remains to be seen.

Bob: If so, we'll have to do some very serious speed-up. My impression so far has been that Ruby is at least a couple orders of magnitude slower than the equivalent C or Fortran implementation.

Alice: In addition, our leapfrog calculates the acceleration twice on the `Nbody` level, and for each particle pair, the relative acceleration is also computed twice.

Bob: If we had been a little more clever, we would have saved a factor of four there too. I bet we can speed up our code by a stunning factor of a thousand or so, if we pull all stops!

Alice: Maybe, we'll see in due time. For now, I think we *are* being clever, by not worrying at all about optimization. The point is to bring out the underlying structure, which is complex enough all by itself. Once we really see that clearly, we can start optimizing while avoiding confusing clutter.

2.4 A Matter of Taste

Bob: Note, by the way, one more difference between the 2-body and N-body case: in the latter case we have to accumulate the results, through the summation you just showed. Before traversing the loop over particles, we have to clear the vector where the acceleration `a` on our particle is being stored. I experimented with various ways to do so, but the most compact notation I found was what I wrote on the top of the `acc` method:

```
a = @pos*0                    # null vector of the correct length
```

Isn't that a nifty and compact expression?

Alice: I see. In order to provide a null vector for the acceleration with the right number of components, you use the position as a template, and after copying the position, you fill all entries with zeroes. I'm glad you put a comment line in, since otherwise the meaning wouldn't have been so obvious at first reading.

Hmmm. While I agree that it is compact, perhaps a longer expression would have been a bit more clear. How about

```
a = ([0]*@pos.size).to_v           # null vector of the correct length
```

Bob: Yes, that would bring out the fact that you use the position vector *only* because you want to extract its size, and not for any other reason. And you explicitly show how you start with an array of length 1, filled with a single 0, and then extend that array to contain `@pos.size` components. But then you still have to convert it into a vector. You see, I avoided the last step by starting with a copy of `@pos`, which was already a vector.

Alice: Yes, your construction was clever, but I'm still wondering about the unsuspecting reader, who has to make sense of your cleverness. In fact, in my more lengthy alternative, notice that I left your comment line in, since upon first reading, even my longer line would still not be fully clear, I'm afraid.

If I really wanted to be self-explanatory, I would write:

```
vector_size = @pos.size
a = ([0]*vector_size).to_v
```

That way I would express the fact that `a` is a vector, that it needs to be of the right size, that it should contain all zeroes, and that it should be converted to a proper vector at the end of the day.

Bob: A long day, if you ask me. I would never have guessed your explanation completely just from looking at that piece of code, so I would insert a comment there as well – which makes your alternative longer than mine. I prefer to stick with my

```
    a = @pos*0                     # null vector of the correct length
```

Alice: Fine with me. This is really a matter of taste.

2.5 Potential Energy

Bob: While we're at it, let me walk you through the rest of the `Body` class definition. The potential is constructed in a very similar way as the acceleration, by doing a body walk through the whole system. In the two-body case, we started with:

```
  def epot                         # potential energy
    -@mass/sqrt(@pos*@pos)         # per unit of reduced mass
  end
```

My new N-body version reads:

```
  def epot                         # potential energy
    p = 0
    @nb.body.each do |b|
      unless b == self
        r = b.pos - @pos
        p += -@mass*b.mass/sqrt(r*r)
      end
    end
    p
  end
```

The difference with respect to `acc` is that the potential energy includes a product of the mass of the calling particle i and the mass of the called particle j:

$$E_{pot,i} = -G \sum_{\substack{j=1 \\ j \neq i}}^{N} \frac{M_i M_j}{|\mathbf{r}_j - \mathbf{r}_i|} \tag{2.3}$$

Alice: Yes, of course, and that is something that is easy to leave out. If I would have written the potential method `epot`, using the acceleration method `acc` as a template, I might have forgotten to include the factor `@mass` for the calling particle. And such a bug might be hard to find at first, since we tend to test a code with simple values for the masses, often just unity. In general, it is important to run tests with masses that are not all unity.

2.6 Local Arrays

Bob: The rest of the input and output routines are unchanged, compared to our earlier two-body code. Let's return to the `Nbody` class. You mentioned that the `leapfrog` method was almost the same as before. Unfortunately, that is the only one of our four integration methods that remained quite simple to read. The other three have become a bit more crowded, I'm afraid.

Alice: I'll start with the forward Euler case again. In `forward`, you have replaced the previous form

```
def forward(dt)
  old_acc = acc
  @pos += @vel*dt
  @vel += old_acc*dt
end
```

by:

```
def forward(dt)
  old_acc = []
  @body.each_index{|i| old_acc[i] = @body[i].acc}
  @body.each{|b| b.pos += b.vel*dt}
  @body.each_index{|i| @body[i].vel += old_acc[i]*dt}
end
```

Bob: This is a tricky point. Before we stored the original acceleration in the vector `old_acc`, which was a single physical vector, containing the relative acceleration between our two particles. In the N-dimensional analogue that we have here, we need to store N initial acceleration vectors, one for each particle.

The most straightforward solution would be to define a new instance variable for the `Body` class, `@old_acc`, but I rejected that solution. As you will be happy to hear, I wanted to keep the code modular, without letting the `Body` class know what the `Nbody` class might decide to be good algorithms. The alternative would be to saddle the poor `Body` class simultaneously with all the possible variables that would be needed in all the algorithms you could choose from.

Alice: I indeed applaud your desire for modularity. However, in this particular case I'm not so sure whether we should insist on such a strong separation. Let's get back to that in a moment.

Bob: Since I wanted to keep the auxiliary variables, such as `old_acc`, local, I could not loop over them using the `@body.each` construct used in `leapfrog`. Of course, you can loop over `old_acc` alone easily enough, in a `old_acc.each` construction, but that would in turn not allow the `@body` array to be traversed.

The only solution I saw was to introduce an index i – yes, I know, we just celebrated the lack of indices i and j in Ruby, and I'm not happy with it, but at least for now, it works. In that way, I could use the `Array` method `@body.each_index`, which does what it says it does, namely traversing the `@body` array. Now since `old_acc` and `@body` have the same number of components, equal to the number of particles N, this one construct can simultaneously traverse both arrays. The i index is the glue that connects both traversals, keeping them in lock step.

In addition, I had to introduce the array `old_acc`, which I did here in the first line of `forward`. The third line did not contain any local variables, so there at least I could avoid the use of an i variable.

Alice: That is a reasonable solution. You are trading modularity for readability. And while I'm sure there are several alternatives, let's first complete our guided tour here. What is left to visit is the energy diagnostics part of the code.

2.7 Energy Diagnostics

Bob: That turned out to be really simple. For each `Body` method `ekin` there is a corresponding `Nbody` method `ekin` that gathers all the individual results, and sums them up to find the total kinetic energy. Here is the `Body` version:

```
def ekin                              # kinetic energy
  0.5*(@vel*@vel)
end
```

and here is the `Nbody` version:

```
def ekin                              # kinetic energy
  e = 0
  @body.each{|b| e += b.ekin}
  e
end
```

In order to sum it all up, I introduce a variable `e`, initialize it to zero, add the various contributions, and then I list `e` again, in the final line. In that way, the method `ekin` returns the correct value `e`.

Alice: Just curious: couldn't you have left out the last line, with the single `e`? At the last time that the statement in the previous line will be executed, some particle's kinetic energy will be added to `e`, so `e` will be what is going to be returned anyway, no?

Bob: I'm not sure. You're talking about the last action in a loop, and then control is being returned to the `each` method. But it is easy enough to find out whether that would work. Let's call `irb` for help:

```
|gravity> irb
irb(main):001:0> a = [1, 2, 3]
=> [1, 2, 3]
irb(main):002:0> e = 2
=> 2
irb(main):003:0> a.each{|b| e += b}
=> [0, 1, 2, 3]
irb(main):004:0> e
=> 8
```

Alice: I see. You were correct in worrying about the control coming back to the array. Actually, that makes sense: it was the array `a` in this example that called the each method. And frankly, even if it would have worked, it might have been better to leave the final `e` line in there, at the end of your `ekin`, for clarity.

Bob: Well, I wasn't sure either. Learn something new everyday. And it is certainly nice to work with an interpreted, rather than a compiled language: this type of checking you can do extremely quickly and easily.

Alice: The story for the potential energy must be similar. For each particle we have a method `epot` associated with the `Body` class, as you just showed us already, and a method with the same name, but associated with the `Nbody` class:

```
def epot                          # potential energy
  e = 0
  @body.each{|b| e += b.epot}
  e/2                             # pairwise potentials were counted twice
end
```

Just like for the kinetic energy, the `Nbody` method `epot` gathers all the contributions to the potential energy of the various bodies – with one twist: you are now dividing by a factor two. Ah, of course: for each particle pair, the contribution is counted once when the one particle computes its potential energy, and once again when the other particle computes its potential energy. Therefore, every particle pair contribution gets counted twice, and at the end we have to correct for that.

Bob: Yes, indeed. And yes, I had left that factor of two out, the first time I ran the program. Diagnostics are wonderful; they sure keep you honest: of course I could get no good energy conservation no matter what I tried, until I realized what was going wrong. I found it by computing the initial kinetic energy and potential energy for a two-body system, which was easy enough to do on paper. Comparing it with the numerical result, it was immediately clear that the potential energy was counted twice.

Chapter 3

Testing the N-body Code

3.1 A 2-Body System

Alice: I would like to see the N-body program running for a 2-body system first, just to check whether we really get the same results.

Bob: That's a good idea. I have tested it so far with a 3-body system, with some randomly chosen initial conditions, but I agree that it would be good to test the code from the ground up. Shall we try to reproduce the same Kepler orbit that we integrated using our `euler.in` initial conditions? They were

```
1
1 0
0 0.5
```

Alice: Yes, but now we have to be careful how we interpret this mass value of 1 that we used before. Remember how we introduced the two-body problem, using the relative position $\mathbf{r}$ between the two pairs? The equation of motion for $\mathbf{r}$ was

$$\frac{d^2}{dt^2}\mathbf{r} = -G\frac{M_1 + M_2}{r^3}\mathbf{r} \tag{3.1}$$

So all we know is that the sum of the masses is unity. We can divide this over the two particles in any way we like. We could take them to be of equal mass, in which case $M_1 = M_2 = \frac{1}{2}$. However, I would prefer unequal masses, just to avoid degenerate situations where our test may fail to uncover some subtle bug.

Bob: Good idea. We saw already how using a mass of unity could fail to show an error in mass assignment. The more asymmetric and non-default our choice is, the better. It would be good, though, to calculate the orbits in the center-of-mass frame, otherwise the results are more difficult to interpret, when the particles start drifting off, away from the origin. How about this choice? I'll put it in a file `test1.in`:

```
2
0
0.8
0.2  0
0    0.1
0.2
-0.8  0
0    -0.4
```

I will first redo the fourth-order Runge-Kutta run for a time step of 10^{-4}, using our previous two-body code, `integrator_driver2h.rb`:

```
require "rkbody.rb"

include Math

dt = 0.0001          # time step
dt_dia = 10          # diagnostics printing interval
dt_out = 10          # output interval
dt_end = 10          # duration of the integration
method = "rk4"       # integration method

STDERR.print "dt = ", dt, "\n",
      "dt_dia = ", dt_dia, "\n",
      "dt_out = ", dt_out, "\n",
      "dt_end = ", dt_end, "\n",
      "method = ", method, "\n"

b = Body.new
b.simple_read
b.evolve(method, dt, dt_dia, dt_out, dt_end)
```

Here is the result:

```
|gravity> ruby integrator_driver2h.rb < euler.in
dt = 0.0001
dt_dia = 10
dt_out = 10
dt_end = 10
method = rk4
```

```
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 100000 steps :
  E_kin = 0.554 , E_pot =  -1.43 , E_tot = -0.875
             E_tot - E_init = -8.33e-14
  (E_tot - E_init) / E_init =9.52e-14
  1.0000000000000000e+00
  5.9961755488723312e-01 -3.6063458344261029e-01
  1.0308069102701605e+00  2.1389530419780176e-01
```

3.2 A Bug

Now let's see what my new N-body code, `rknbody1a_driver.rb`, will do. Here is the code:

```
require "rknbody1.rb"

include Math

dt = 0.0001          # time step
dt_dia = 10          # diagnostics printing interval
dt_out = 10          # output interval
dt_end = 10          # duration of the integration
method = "rk4"       # integration method

STDERR.print "dt = ", dt, "\n",
      "dt_dia = ", dt_dia, "\n",
      "dt_out = ", dt_out, "\n",
      "dt_end = ", dt_end, "\n",
      "method = ", method, "\n"

nb = Nbody.new
nb.simple_read
nb.evolve(method, dt, dt_dia, dt_out, dt_end)
```

And here is what it does:

```
|gravity> ruby rknbody1a_driver.rb < test1.in
dt = 0.0001
dt_dia = 10
```

```
dt_out = 10
dt_end = 10
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0.085 , E_pot =  -0.16 , E_tot = -0.075
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 10, after 100000 steps :
  E_kin = 0.377 , E_pot =  -0.229 , E_tot = 0.148
             E_tot - E_init = 0.223
  (E_tot - E_init) / E_init = -2.98
2
  9.9999999999900329e+00
  8.0000000000000004e-01
  1.1992351097726084e-01 -7.2126916688572407e-02
  2.0616138205436191e-01  4.2779060839347856e-02
  2.0000000000000001e-01
 -4.7969404390904336e-01  2.8850766675428963e-01
 -8.2464552821744763e-01 -1.7111624335739142e-01
```

Alice: Huh? An energy conservation error of order unity? And our old code was conserving energy almost on machine accuracy! Are you sure you have tested your code?

Bob: Yes, I'm sure, I can show you. And yes, I'm deeply puzzled now.

3.3 The Simplest Case

Alice: Let's try a very simple situation, where we absolutely know what the outcome will be, in explicit form. Let us take a circular binary star with equal masses, just to see what will go wrong there. Perhaps that will give us a hint. We can give both stars a mass $m_1 = m_2 = 1$, and start with an initial distance of $r_{i,j} = 1$. This gives us an initial potential energy of $-m_1m_2/r_{i,j} = -1$, since we are working with $G = 1$. Because of the virial theorem, we know that the average kinetic energy has to be $-1/2$ times that of the average potential energy. In a circular binary, both kinetic and potential energies remain constant, and equal to their initial values, and therefore the total initial kinetic energy is $1/2$, in the center-of-mass frame. This means that we need to give each star a kinetic energy of $1/4$. Since each star has a mass of 1, the velocity v of each star should be $1/\sqrt(2)$, in order to make $E_{kin} = \frac{1}{2}mv^2 = 1/4$ for that star.

Bob: Here we go. I will call the initial file for the circular binary `test2.in`.

```
2
```

```
0
1
0.5  0
0    7.071067811865475e-01
1
-0.5  0
0    -7.071067811865475e-01
```

I'll use the same parameters for the fourth-order Runge-Kutta integrator, in my N-body code:

```
|gravity> ruby rknbody1a_driver.rb < test2.in
dt = 0.0001
dt_dia = 10
dt_out = 10
dt_end = 10
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0.5 , E_pot =  -1 , E_tot = -0.5
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 10, after 100000 steps :
  E_kin = 0.5 , E_pot =  -1 , E_tot = -0.5
             E_tot - E_init = -6.22e-15
  (E_tot - E_init) / E_init = 1.24e-14
2
  9.9999999999900329e+00
  1.0000000000000000e+00
 -2.4843310663498000e-03  4.9999382806106440e-01
 -7.0709805274678161e-01 -3.5133746874538431e-03
  1.0000000000000000e+00
  2.4843310663498000e-03 -4.9999382806106440e-01
  7.0709805274678161e-01  3.5133746874538431e-03
```

3.4 A Variation

Alice: Congratulations! You *do* have a working integrator, at least for a circular equal-mass binary. But of course the question remains: what went wrong with the non-circular non-equal-mass binary?

Bob: I'm stumped. But this is a bug we should be able to track down without too much trouble. The last case, which worked, was special in at least three ways: the orbit was circular, the masses were equal, and the masses were also all equal to unity. The case which didn't work did have none of these three idealizations. Let's modify each of those in turn.

Alice: It may be easiest to drop the unity of the masses. If we make the mass ten times smaller, the potential energy becomes one hundred times smaller, and so should the kinetic energy of each particle. Since the mass is already ten times smaller, we can make the kinetic energy a hundred times smaller but lowering the velocity by a factor $1/\sqrt(10)$. This means that the new velocity of each particle should become $v = (1/\sqrt(2))(1/\sqrt(10)) = 1/2\sqrt(5)$.

Bob: Here goes, with `test3.in`:

```
2
0
0.1
0.5  0
0    0.22360679774997896964
0.1
-0.5  0
0    -0.22360679774997896964
```

I'll use the same parameters for the fourth-order Runge-Kutta integrator, in my N-body code:

```
|gravity> ruby rknbody1a_driver.rb < test3.in
dt = 0.0001
dt_dia = 10
dt_out = 10
dt_end = 10
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0.05 , E_pot =  -0.01 , E_tot = 0.04
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = 0
at time t = 10, after 100000 steps :
  E_kin = 0.05 , E_pot =  -0.01 , E_tot = 0.04
             E_tot - E_init = -2.64e-15
  (E_tot - E_init) / E_init = -6.59e-14
2
  9.9999999999900329e+00
  1.0000000000000001e-01
 -1.1897419599036606e-01 -4.8563889948034655e-01
  2.1718431835122404e-01 -5.3206877960568291e-02
  1.0000000000000001e-01
  1.1897419599036606e-01  4.8563889948034655e-01
 -2.1718431835122404e-01  5.3206877960568291e-02
```

3.5 Another Variation

Alice: Nothing wrong here. So changing the masses did not help, at least not for our circular orbit. Shall we try to increase the eccentricity, while leaving the masses both unity? We can just make the velocities a bit smaller. How about this, as the file `test4.in`:

```
2
0
1
0.5  0
0    0.5
1
-0.5  0
0    -0.5
```

Here goes:

```
|gravity> ruby rknbody1a_driver.rb < test4.in
dt = 0.0001
dt_dia = 10
dt_out = 10
dt_end = 10
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0.25 , E_pot =  -1 , E_tot = -0.75
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 10, after 100000 steps :
  E_kin = 0.307 , E_pot =  -1.06 , E_tot = -0.75
             E_tot - E_init = -1.35e-14
  (E_tot - E_init) / E_init = 1.81e-14
2
  9.9999999999900329e+00
  1.0000000000000000e+00
  4.4625642676571020e-01  1.5717985834439904e-01
 -3.3221408890524584e-01  4.4320400716535185e-01
  1.0000000000000000e+00
 -4.4625642676571020e-01 -1.5717985834439904e-01
  3.3221408890524584e-01 -4.4320400716535185e-01
```

3.6 Two Variations

Bob: Still no cigar. Nothing wrong here either. How about changing both the masses and the eccentricity? I'll just make the masses ten percent smaller, while leaving everything else the same, calling the file `test5.in`:

```
2
0
0.9
0.5  0
0    0.5
0.9
-0.5  0
0    -0.5
```

Try again:

```
|gravity> ruby rknbody1a_driver.rb < test5.in
dt = 0.0001
dt_dia = 10
dt_out = 10
dt_end = 10
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0.25 , E_pot =  -0.81 , E_tot = -0.56
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 10, after 100000 steps :
  E_kin = 0.56 , E_pot =  -1.09 , E_tot = -0.529
             E_tot - E_init = 0.031
  (E_tot - E_init) / E_init = -0.0554
2
  9.9999999999900329e+00
  9.0000000000000002e-01
  2.1147553247493753e-01 -3.0575926734969655e-01
  7.4020397769574453e-01  1.1195514589010475e-01
  9.0000000000000002e-01
 -2.1147553247493753e-01  3.0575926734969655e-01
 -7.4020397769574453e-01 -1.1195514589010475e-01
```

Alice: Here we clearly have a problem, and a big one: terrible energy conservation. So now we know that the problem does not depend on having unequal masses, but it does seem to require both non-circularity and masses that differ from unity.

Bob: We're getting a little closer, but we may still have quite a ways to go!

3.7 A Single Time Step

Alice: Let's take a single time step, to see where and how things go wrong. And let's take a relatively big step. That should make it easier to interpret the output:

```
|gravity> ruby rknbody1b_driver.rb < test5.in
dt = 0.01
dt_dia = 0.01
dt_out = 0.01
dt_end = 0.01
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0.25 , E_pot =  -0.81 , E_tot = -0.56
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 0.01, after 1 steps :
  E_kin = 0.25 , E_pot =  -0.81 , E_tot = -0.56
             E_tot - E_init = 3.6e-06
  (E_tot - E_init) / E_init = -6.43e-06
2
  1.0000000000000000e-02
  9.0000000000000002e-01
  4.9995499977502056e-01  4.9998499955000017e-03
 -9.0000900017438174e-03  4.9995499797487092e-01
  9.0000000000000002e-01
 -4.9995499977502056e-01 -4.9998499955000017e-03
  9.0000900017438174e-03 -4.9995499797487092e-01
```

Bob: But not nearly easy enough! I don't like to compute by hand all the steps in the fourth-order Runge-Kutta algorithm. How about checking whether the forward Euler breaks down as well? That will be far easier to check by hand.

Alice: You may be right. Let's take a break first, and then take a fresh look at the whole situation.

Chapter 4

Debugging the N-body Code

4.1 Forward Euler

Bob: Okay, ready to chase and catch our bug?

Alice: Sure thing! And I like your idea to simplify things and turn to our old friend, the forward Euler algorithm. And we may as well make the total integration time a bit shorter, to speed things up.

```
|gravity> ruby rknbody1c_driver.rb < test5.in
dt = 0.0001
dt_dia = 1
dt_out = 1
dt_end = 1
method = forward
at time t = 0, after 0 steps :
  E_kin = 0.25 , E_pot =  -0.81 , E_tot = -0.56
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 1, after 10000 steps :
  E_kin = 0.875 , E_pot =  -1.37 , E_tot = -0.497
             E_tot - E_init = 0.0628
  (E_tot - E_init) / E_init = -0.112
2
  9.9999999999990619e-01
  9.0000000000000002e-01
  3.8764717723091237e-02  2.9255040141287891e-01
 -8.9213055091367810e-01 -2.8150072109002572e-01
```

```
  9.0000000000000002e-01
 -3.8764717723091237e-02 -2.9255040141287891e-01
  8.9213055091367810e-01  2.8150072109002572e-01
```

Bob: That doesn't look too good, but then again, forward Euler is not a great integrator. I'll give it a ten times smaller time step:

```
|gravity> ruby rknbody1d_driver.rb < test5.in
dt = 1.0e-05
dt_dia = 1
dt_out = 1
dt_end = 1
method = forward
at time t = 0, after 0 steps :
  E_kin = 0.25 , E_pot =  -0.81 , E_tot = -0.56
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 1, after 100000 steps :
  E_kin = 0.876 , E_pot =  -1.37 , E_tot = -0.497
             E_tot - E_init = 0.0626
  (E_tot - E_init) / E_init = -0.112
2
  9.9999999999808376e-01
  9.0000000000000002e-01
  3.8685902921474627e-02  2.9243384480372231e-01
 -8.9221956236710909e-01 -2.8194011853141043e-01
  9.0000000000000002e-01
 -3.8685902921474627e-02 -2.9243384480372231e-01
  8.9221956236710909e-01  2.8194011853141043e-01
```

4.2 Nothing Wrong

Alice: Okay, this is really wrong. Almost the same magnitude for the error.

Bob: Now that we know that the forward Euler method also fails for this orbit, we can take a single time step, and track things by hand.

```
|gravity> ruby rknbody1e_driver.rb < test5.in
dt = 0.01
dt_dia = 0.01
dt_out = 0.01
dt_end = 0.01
```

```
method = forward
at time t = 0, after 0 steps :
  E_kin = 0.25 , E_pot =  -0.81 , E_tot = -0.56
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 0.01, after 1 steps :
  E_kin = 0.25 , E_pot =  -0.81 , E_tot = -0.56
             E_tot - E_init = 0.000121
  (E_tot - E_init) / E_init = -0.000217
2
  1.0000000000000000e-02
  9.0000000000000002e-01
  5.0000000000000000e-01  5.0000000000000001e-03
 -9.0000000000000011e-03  5.0000000000000000e-01
  9.0000000000000002e-01
 -5.0000000000000000e-01 -5.0000000000000001e-03
  9.0000000000000011e-03 -5.0000000000000000e-01
```

Let me print out the initial conditions again:

```
2
0
0.9
0.5  0
0    0.5
0.9
-0.5  0
0    -0.5
```

The initial velocity is along the y axis, so after one step the x components of the positions of the particles should not be affected. Indeed, they remain at 0.5.

The y components of the positions should be equal to $\mathbf{v}\Delta t = 0.01\mathbf{v}$. And so they are.

As for the velocities, their y components should be unchanged, and indeed that is the case.

Finally, the x components of the velocities should be equal to $\mathbf{a}\Delta t = 0.01\mathbf{a}$. The acceleration, for particles of mass 0.9, at distance 1, should be 0.9, which means that $|\mathbf{a}\Delta t| = 0.009$. All that checks too.

Alice: Yes, it is much easier to visually inspect a forward Euler scheme than a fourth-order Runge-Kutta, I agree!

Bob: But how strange. It seems that there is really nothing wrong with this step. If this step is correct, how can things go wrong later on? Admittedly, we are still starting from a somewhat special case, launching our particles parallel to the y axis, while being positioned on the x axis. Perhaps we should take a really generic initial position, not lined up with anything at all?

4.3 Two Possibilities

Alice: Hmmm. You know, it might indeed be that the integration is proceeding fine, but that there is an error in the determination of the energy error.

Bob: Ah, that could well be the case. That would be like trying to land an airplane, and to see a warning light coming on, telling you that your landing gear is not fully unfolded. There are two possibilities: either your landing gear is faulty or the warning light is malfunctioning.

Alice: If I were the pilot, I would surely hope that it is the warning light that is at fault. In our case either way is no problem, either way, as long as we can trace where what went wrong.

Bob: But for tracking down energy diagnostics, we really have to get back to the code, and read all the lines that compute energies. Unlike the integration itself, where we can slow down to take just one time step, the energy diagnostics have no free parameter; either you do it or you don't.

Alice: Before we look at the code, let us stare at the output just a bit longer. It still may give us a clue. But I don't like to think about slightly eccentric and slightly non-unity masses. Let us run the equal-mass circular binary with masses unity once more, and that one, too, for just one time step:

```
|gravity> ruby rknbody1e_driver.rb < test2.in
dt = 0.01
dt_dia = 0.01
dt_out = 0.01
dt_end = 0.01
method = forward
at time t = 0, after 0 steps :
  E_kin = 0.5 , E_pot =  -1 , E_tot = -0.5
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 0.01, after 1 steps :
  E_kin = 0.5 , E_pot =  -1 , E_tot = -0.5
             E_tot - E_init = 0.0002
  (E_tot - E_init) / E_init = -0.0004
2
  1.0000000000000000e-02
  1.0000000000000000e+00
  5.0000000000000000e-01  7.0710678118654745e-03
```

```
 -1.0000000000000000e-02  7.0710678118654746e-01
  1.0000000000000000e+00
 -5.0000000000000000e-01 -7.0710678118654745e-03
  1.0000000000000000e-02 -7.0710678118654746e-01
```

Bob: This one, too, is perfect. I'm getting better at 'reading' the forward Euler output: I can see now immediately that both the directions and the magnitudes of the increments in position and velocity are correct. The acceleration here is just 1.

4.4 Back to Square One

Alice: I guess we'll have to walk through the code, much as I don't like to do that, as a matter of principle. A good code should give you enough diagnostics to allow you to track down a bug by treating it as a black box.

Bob: The only other numbers here, apart from the energy errors, are the kinetic and potential energy, and they are obviously correct, given the values for the circular binary: at distance 1 and masses 1, the potential energy must be -1, and with velocity 0.5, each kinetic energy is 1/4, so 0.5 in total.

Alice: Ah, you are looking at the energies at time `t = 0`. That is a great idea: if it is a matter of the warning light malfunctioning, chances are that it already malfunctioned when the plane took off!

Bob: But I've just shown that it did *not* malfunction!

Alice: But the circular binary did not give us any problems. It was the non-unity masses and the eccentricity that did it. Let us redo that one forward Euler step, which showed a correct integration in that case, and let us check by hand whether the initial energy is computed correctly there. Let us put everything on the table once more. First the initial conditions:

```
2
0
0.9
0.5  0
0    0.5
0.9
-0.5  0
0    -0.5
```

Then the result of the one time step:

```
|gravity> ruby rknbody1e_driver.rb < test5.in
dt = 0.01
dt_dia = 0.01
dt_out = 0.01
dt_end = 0.01
method = forward
at time t = 0, after 0 steps :
  E_kin = 0.25 , E_pot =  -0.81 , E_tot = -0.56
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 0.01, after 1 steps :
  E_kin = 0.25 , E_pot =  -0.81 , E_tot = -0.56
             E_tot - E_init = 0.000121
  (E_tot - E_init) / E_init = -0.000217
2
  1.0000000000000000e-02
  9.0000000000000002e-01
  5.0000000000000000e-01  5.0000000000000001e-03
 -9.0000000000000011e-03  5.0000000000000000e-01
  9.0000000000000002e-01
 -5.0000000000000000e-01 -5.0000000000000001e-03
  9.0000000000000011e-03 -5.0000000000000000e-01
```

The initial potential energy must be equal to the square of the masses, divided by the distance, $0.9 \times 0.9/1 = 0.81$. That is indeed what the code gives. The initial kinetic energy for each particle must be $0.5 \times 0.9 \times 0.5 \times 0.5 = 0.1125$, so for both particles together 0.225. Hey! The code output gives us 0.25 !

4.5 A Warning Light

Bob: So it is the warning light, after all. The potential energy gets calculated correctly, but the kinetic energy doesn't. That will be easy to check. Here is the method in the `Body` class:

```
  def ekin                         # kinetic energy
    0.5*(@vel*@vel)
  end
```

And indeed, I left out the mass! How simple. Of course, it should have been:

```
  def ekin                         # kinetic energy
    0.5*@mass*(@vel*@vel)
  end
```

And I can understand now why I made that mistake: in our earlier two-body code the corresponding method was:

```
def ekin                          # kinetic energy
  0.5*(@vel*@vel)                 # per unit of reduced mass
end
```

and as the comment indicated, there it was defined *per unit of reduced mass.* I just erased the comment, since I knew that the concept of reduced mass only applies to a two-body problem, and not to the general N-body problem. But although I erased the comment, I failed to change the code line itself, by adding the mass factor!

What a blunder. And I even commented on the fact that I was so careful to include an extra mass factor in my definition of the potential energy! But in that case, I was modeling the potential energy method `epot` after the method `acc` that calculates the acceleration, because both involve a loop over all other particles. That is why I did not compare the `epot` and `ekin` methods.

Alice: Well, it is an easy mistake to make. I have made plenty of much more obvious mistakes in my life! The challenge is not so much to write correct code, but to debug code correctly. And I think we did pretty well, given the fact that this was a very tricky bug to discover in the first place.

Bob: Tricky indeed: if we would have only tried a circular orbit, even with masses that were not unity, we would have never found this bug. I'm very glad now that you insisted on using unequal masses!

Alice: Unequal masses and eccentricity. If we would have used unequal masses for an eccentric orbit, we *still* would not have seen a problem with energy conservation. The energy would have been calculated wrong at time `t=0`, but for a circular orbit, energy and potential remain constant throughout the orbit. So the same mistake in kinetic energy would have been present at any later time, and the code would have reported no significant energy change; our energy drift warning light would not have come on!

Bob: You are right. Now that is a tricky bug. Clearly the moral of the story is: test any new code for generic input data, not only for input data that are easy to generate and easy to interpret.

4.6 Hindsight

Alice: Ah, but wait a minute! Before we congratulate ourselves too much, I wonder whether we should not have noticed that there was something wrong with the energies in the case of the circular orbit where the masses of the two stars were equal to each other, but different from unity. Let us look at that case again. Where did we file those data? Ah, here it is: we stored the initial conditions in the file `test3.in`:

```
2
0
0.1
0.5  0
0    0.22360679774997896964
0.1
-0.5  0
0    -0.22360679774997896964
```

Let us run that case again, with the original code `rknbody1.rb`:

```
|gravity> ruby rknbody1a_driver.rb < test3.in
dt = 0.0001
dt_dia = 10
dt_out = 10
dt_end = 10
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0.05 , E_pot =  -0.01 , E_tot = 0.04
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = 0
at time t = 10, after 100000 steps :
  E_kin = 0.05 , E_pot =  -0.01 , E_tot = 0.04
             E_tot - E_init = -2.64e-15
  (E_tot - E_init) / E_init = -6.59e-14
2
  9.9999999999900329e+00
  1.0000000000000001e-01
 -1.1897419599036606e-01 -4.8563889948034655e-01
  2.1718431835122404e-01 -5.3206877960568291e-02
  1.0000000000000001e-01
  1.1897419599036606e-01  4.8563889948034655e-01
 -2.1718431835122404e-01  5.3206877960568291e-02
```

Bob: You are right! Now that we know what to look for, it is obvious that the diagnostics output is totally wrong. Already at time `t = 0`, the total energy is positive: `E_tot = 0.04`. A binary with positive total energy will immediately fly apart. We should have noticed that right away!

Alice: We should have, yes, but we didn't. It goes to show: even with good diagnostics, it is easy to overlook a trouble signal. To continue with your analogy of a pilot, looking at a warning signal: in a cockpit with many different indicators, one can overlook the fact that one of the meters gives an impossible result. We were staring at the red light of errors in energy conservation, and we overlooked the less obvious lights.

Bob: Still, I feel pretty stupid that I didn't see that we were setting up a binary with binding energy of the wrong sign.

Alice: Well, that makes two of us. But the goal of good software development is not to make programmers perfect, since that is impossible. Instead, the goal should be to make debugging easy enough, so that imperfect people can still converge to an almost-perfect code, when they try hard enough. This in itself is already enough of a reason for using a language like Ruby, where the programmers are not bogged down with trying to keep the compiler happy. We can at least spend all of our energy on the debugging process itself, without worrying about declarations and type checking and all the other constraints that the more classical languages always bring with them.

Bob: You have a charitable interpretation of human weakness. Even so, I really will try not to make this kind of silly mistake again.

Alice: Famous last words!

4.7 Bug Fixed

Bob: So, we're done!

Alice: It seems that way. But it would not hurt to test our original run again, now that you have fixed the code. You know, it would not be the first time that a code contains more than one bug. Another common mistake people make is to find a glaring bug in a code, fix it, and then happily declare the code to be correct, without looking further.

Bob: Okay! I will call the corrected version `rknbody2.rb`, since I would like to keep the original file `rknbody1.rb`, to show to my students later on. It would be interesting to see how they would go about debugging it. Here is our original run, but now with the corrected code:

```
|gravity> ruby rknbody2a_driver.rb < test1.in
dt = 0.0001
dt_dia = 10
dt_out = 10
dt_end = 10
method = rk4
at time t = 0, after 0 steps :
```

```
  E_kin = 0.02 , E_pot =  -0.16 , E_tot = -0.14
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 10, after 100000 steps :
  E_kin = 0.0887 , E_pot =  -0.229 , E_tot = -0.14
             E_tot - E_init = 4.55e-15
  (E_tot - E_init) / E_init = -3.25e-14
2
  9.9999999999900329e+00
  8.0000000000000004e-01
  1.1992351097726084e-01 -7.2126916688572407e-02
  2.0616138205436191e-01  4.2779060839347856e-02
  2.0000000000000001e-01
 -4.7969404390904336e-01  2.8850766675428963e-01
 -8.2464552821744763e-01 -1.7111624335739142e-01
```

4.8 One More Check

Alice: That's it: the energy conservation is perfect. But let us not declare victory too early: let's compare the actual positions and velocities with what we got with our earlier two-body code. However, to compare the data, it would be better to convert our results to relative coordinates between the two particles. That should be easy: let us make a little script for this simple form of data reduction, and call it `rknbody2a_reduce.rb`:

```
require "rknbody2.rb"

include Math

nb = Nbody.new
nb.simple_read

relative_position = nb.body[0].pos - nb.body[1].pos
relative_velocity = nb.body[0].vel - nb.body[1].vel

print "relative_position = [", relative_position[0], ", ",
      relative_position[1], "]\n"
print "relative_velocity = [", relative_velocity[0], ", ",
      relative_velocity[1], "]\n"
```

All it does is to read in an N-body system, and print the relative positions and velocities of the first two particles with respect to each other. We can then pipe the results from our previous calculation into our new script:

```
|gravity> ruby rknbody2a_driver.rb < test1.in | ruby rknbody2a_reduce.rb
dt = 0.0001
dt_dia = 10
dt_out = 10
dt_end = 10
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0.02 , E_pot =  -0.16 , E_tot = -0.14
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 10, after 100000 steps :
  E_kin = 0.0887 , E_pot =  -0.229 , E_tot = -0.14
             E_tot - E_init = 4.55e-15
  (E_tot - E_init) / E_init = -3.25e-14
relative_position = [0.599617554886304, -0.360634583442862]
relative_velocity = [1.03080691027181, 0.213895304196739]
```

And compare the results with what we found directly with our two-body code:

```
|gravity> ruby integrator_driver2h.rb < euler.in
dt = 0.0001
dt_dia = 10
dt_out = 10
dt_end = 10
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0.125 , E_pot =  -1 , E_tot = -0.875
             E_tot - E_init = 0
  (E_tot - E_init) / E_init =-0
at time t = 10, after 100000 steps :
  E_kin = 0.554 , E_pot =  -1.43 , E_tot = -0.875
             E_tot - E_init = -8.33e-14
  (E_tot - E_init) / E_init =9.52e-14
  1.0000000000000000e+00
  5.9961755488723312e-01 -3.6063458344261029e-01
  1.0308069102701605e+00  2.1389530419780176e-01
```

Bob: Indeed, very closely the same, to well over ten decimals! Very nice. I think we can declare victory now.

Alice: For the time being, yes. But I would never discount the possibility that somewhere, in some dark corner, some other little bug may still be hiding. However, at some point you just have to move on, while staying vigilant, always being prepared to go back to inspect older codes that you were convinced to be fully debugged.

Bob: By the way, I'm glad we decided to put the energy diagnostics on the standard error stream `STDERR`, while keeping only the particle output on the standard output stream. If we would have written everything on the standard output, it would have been impossible to pipe the data into the next program, as we did above.

Alice: For now, yes, that was a good strategy. But I worry about the idea of scattering related data in completely different directions. Actually, it would be even better to encapsulate the error diagnostics somewhere in the output, in order to keep the data together. Soon we should introduce a new data format, once that keeps everything bundled, but in such a way that the next program will have a fixed and known way to find the data it wants.

Bob: You mean a form of self-describing data? Like the FITS format that observers use? Yes, that would be interesting, and it would not be too hard to implement.

Chapter 5

A Single-Links Version

5.1 A Figure-8 Triple

Bob: Now that we're pretty sure that we can integrate the two-body problem with our new code, how about trying our hand at a three-body system? After all, that was the reason I wrote this program, to go beyond two bodies.

Alice: We will have to choose some initial conditions. Rather than picking something at random, how about trying to integrate a figure-eight configuration?

Bob: You mean this new 'classical' solution to the equal-mass three-body system that was discovered recently by Montgomery and Chenciner? I remember how surprised I was when I first read about that. I had always assumed that the classic celestial mechanicians had found all the interesting solutions already centuries ago, you know, Legendre, Lagrange, Laplace, and perhaps a few others Le's and La's. I was thrilled to see such an elegant new solution.

Alice: Me too. As soon as I read about it, I tried it out for myself, and by golly, the configuration was stable: three stars chasing each other on a figure eight, without the system falling apart. You could even perturb the initial conditions by a fraction of a percent, and preserve stability.

Bob: This all goes to show that we really should implement some form of graphics soon. I'd love to see how the new code will handle that configuration.

Alice: I agree. But at least for now, we can use the figure-eight triple to test the code. I stored the initial conditions somewhere. Ah, here they are. I'll put them in a file named `figure8.in`:

```
3
0
1
0.9700436 -0.24308753
```

```
0.466203685 0.43236573
1
-0.9700436 0.24308753
0.466203685 0.43236573
1
0 0
-0.93240737 -0.86473146
```

The third particle starts in in the center of the figure eight, in the exact center of the coordinate system. The other two bodies move symmetrically with respect to each other, each one third of a period displaced. Ah, my notes tell me that I found the total period of revolution to be about 6.3264 time units. Let us integrate our system for 1/3 of that time: since all particles have the same mass, after 1/3 of the period they should have changed places. That means an integration for a total of 2.1088 time units.

5.2 Switching Places

Bob: Let's predict what will happen. The third particle has velocity components that are negative, both in the x and y direction. This means that it moves to the left and downward. That would suggest that it will in due time reach the position of the second particle, the particle that starts off with a large negative x value. The second particle then has no choice but to replace the first particle, and the first one will in turn wind up in the center. So I predict that an integration of 2.1088 time units will produce the following output:

```
3
0
1
 0        0
-0.93241 -0.86473
1
0.97004  -0.24309
0.46620   0.43237
1
-0.97004  0.24309
 0.46620  0.43237
```

I have only given five digits, since you gave the total time duration to that accuracy, and therefore there is no guarantee that we will halt the calculation exactly after 1/3 of an orbit; in fact we are bound to either overshoot or undershoot, making an error in the fifth or sixth significant digit in all coordinates.

Alice: That seems reasonable. This will be a nice test. Let's try it!

```
|gravity> ruby rknbody2b_driver.rb < figure8.in
dt = 0.001
dt_dia = 2.1088
dt_out = 2.1088
dt_end = 2.1088
method = rk4
at time t = 0, after 0 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2.109, after 2109 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = -2e-15
  (E_tot - E_init) / E_init = 1.55e-15
3
  2.1089999999998787e+00
  1.0000000000000000e+00
 -1.6047303546488470e-04 -1.9320664965417420e-04
 -9.3227640249930266e-01 -8.6473492670753516e-01
  1.0000000000000000e+00
  9.7020367429337440e-01 -2.4296620300772800e-01
  4.6595057278750124e-01  4.3244644507801255e-01
  1.0000000000000000e+00
 -9.7004320125790211e-01  2.4315940965738195e-01
  4.6632582971180025e-01  4.3228848162952316e-01
```

Bob: Not quite as good as I had hoped, but I certainly came close.

Alice: I'd say! And you were right about the order of the particles. The first particle did indeed reach very close to the center, and all other particles reached your predicted position to within a fraction of a tenth of a percent: in all cases the first three digits are correct. Congratulations!

Bob: Thanks! So let's move on.

5.3 Single Links in `Body`

Alice: Well, before we move on, let us revisit the choice you made of installing double links between the particles in an N-body system and their parents. In the `Nbody` class you have an array of bodies `@body` that gives access to the members of the `Body` class, while in the `Body` class there is an instance variable `@nb` that gives each particle access to its parent.

Bob: Yes, I remember that you were not happy with that choice.

Alice: The problem is, I can see the danger of modifying, say, the downward link and forgetting to make a corresponding modification in the upward link. Let's play with some alternatives.

Bob: Be my guest! I'll keep a copy of the last, debugged, version, and it is fun to try out some alternative implementations. After all, Ruby invites this type of prototyping. Here, why don't you take the key board.

Alice: The simplest way to do away with the need for a backward pointer `@nb` would be to give the `Body` method `acc` an extra parameter. Let me take out the declaration of `@nb` in the `attr_accessor` of the `Body` class, and let me replace your `Body` method

```
def acc
  a = @pos*0                         # null vector of the correct length
  @nb.body.each do |b|
    unless b == self
      r = b.pos - @pos
      r2 = r*r
      r3 = r2*sqrt(r2)
      a += r*(b.mass/r3)
    end
  end
  a
end
```

by a new version, that takes the array of bodies as an explicit parameter:

```
def acc(body_array)
  a = @pos*0                                   # null vector of the correct length
  body_array.each do |b|
    unless b == self
      r = b.pos - @pos
      r2 = r*r
      r3 = r2*sqrt(r2)
      a += r*(b.mass/r3)
    end
  end
  a
end
```

Of course, we have to do the same thing for the potential energy calculation, where we replace

```
def epot                        # potential energy
  p = 0
  @nb.body.each do |b|
    unless b == self
      r = b.pos - @pos
      p += -@mass*b.mass/sqrt(r*r)
    end
  end
  p
end
```

by the equivalent version:

```
def epot(body_array)                  # potential energy
  p = 0
  body_array.each do |b|
    unless b == self
      r = b.pos - @pos
      p += -@mass*b.mass/sqrt(r*r)
    end
  end
  p
end
```

5.4 Single Links in `Nbody`

Bob: Okay, and let us put the code into a separate file `rknbody3.rb`. Life gets a bit simpler for the `Nbody` code: when it creates its daughters, it no longer has to tell them who their parent is. Instead of the old version:

```
def initialize(n=0, time = 0)
  @time = time
  @body = []
  for i in 0...n
    @body[i] = Body.new
    @boby[i].nb = self
  end
end
```

we now will use the simpler and more natural version:

```
def initialize(n=0, time = 0)
  @time = time
  @body = []
  for i in 0...n
    @body[i] = Body.new
  end
end
```

And similarly we can leave out the corresponding line in the `simple_read` method: instead of

```
def simple_read
  n = gets.to_i
  @time = gets.to_f
  for i in 0...n
    @body[i] = Body.new
    @body[i].nb = self
    @body[i].simple_read
  end
end
```

we now have only:

```
def simple_read
  n = gets.to_i
  @time = gets.to_f
  for i in 0...n
    @body[i] = Body.new
    @body[i].simple_read
  end
end
```

5.5 The DRY Principle

Alice: Good! I was a bit worried about that repetition. It would have been an easy mistake to make to modify the `initialize` method in one way, and to either forget to modify the `simple_read` method or worse, modify it in a different way. Your original approach violated the DRY principle.

Bob: I wasn't aware of violating anything, let alone a principle I hadn't even heard of. What is the DRY principle?

Alice: Don't Repeat Yourself. The idea is that you should try to avoid stating the same information in more than one place, exactly because it is so easy to update information in one place, and to forget to update it in one or more other places.

Bob: But in many cases it could never hurt you. In C++, for example, you normally declare all your functions, and specify the types of the arguments and the return values, in different places from where you actually define those functions. Now if you make a mistake, and provide conflicting information, the compiler will give you an error message. So there is nothing that can go wrong there.

Alice: Even if there is little danger for errors, it is annoying to have to provide the same information in two places, especially if you work with header files, where the information may even reside in one or more different files. Such requirements go against any notion of rapid prototyping.

While violating the DRY principle may not always lead to likely errors, the consequences are equally bad if it discourages free experimentation. In that way you are likely to miss a more optimal solution. Miss out on more optimal solutions often enough in a large software project, and you get an end product that can be very far from optimal, through the aggregate effect of all the suboptimal decisions you have made.

Bob: You sound like a manager. I must say that I don't like to think in terms of principles, and I tend do develop an allergy against people who follow principles blindly. But I must admit that this particular idea makes a lot of sense. Having to repeat information in different places in C++ is certainly one of the things that I like least about that language. In fact, I've developed an allergy against that feature already quite a while ago.

Alice: I'm glad to hear that your allergy against C++ is stronger than your allergy against me, if I parse you correctly.

Bob: So far, certainly. Let's see how things develop.

Alice: Given that C++ is less likely to change in any fundamental way soon, I guess it's up to us to see how our collaboration develops further. As for me, I think our different attitudes have been balanced quite well, so far.

Bob: I agree. And I'm glad you take my recalcitrance with a sense of humor, since I'm not likely to become 'principled' any time soon!

5.6 Simplifying Further

Alice: This discussion about the DRY principle started when I told you I was glad that the `initialize` method no longer had to repeat what the `simple_read` method also did, namely the initialization of the upward links from particles to N-body system, links that you have now removed. However, I still see some repetition left in the the initialization method, of stuff that is already taken care of in `simple_read`. Here is your latest version of `initialize`:

```
def initialize(n=0, time = 0)
  @time = time
  @body = []
  for i in 0...n
    @body[i] = Body.new
  end
end
```

If you compare this with the input method `simple_read`:

```
def simple_read
  n = gets.to_i
  @time = gets.to_f
  for i in 0...n
    @body[i] = Body.new
    @body[i].simple_read
  end
end
```

you see that the number of particles N and the time are being used here only. After reading in N, `simple_read` extends the array of bodies to contain N elements, and in the process it reads in the values for each body. And after reading in the time, it assigns that value directly to `@time`. It seems to me that we can simplify the `initialize` method to just:

```
def initialize
  @body = []
end
```

Bob: That is a lot shorter, and yes, that would work fine in this case. The reason that I included the two arguments to `initialize` is that I had been thinking about a situation where you may want to construct an initial model, say a Plummer model or a King model. In that case, you probably do want to set up an array of N bodies, at a specified time, before you internally assign values to the masses, positions and velocities of those bodies.

Alice: Ah, yes, in that case these extra arguments would come in handy, I agree. But so far you have often made it clear to me that we should be demand-driven, and that we should not build in extra options for possible future use, unless we actually plan to use those pretty soon.

Bob: I agree. Yes, it would be fine with me to take your simpler version, at least for now. We can easily extend `initialize` back to the longer form later on, when needed.

Alice: great, four lines saved and one line simplified. What else is there left to be done?

5.7 Finishing the Revision

Bob: Next we have to change the way in which `acc` gets called from within `Nbody`. Let us start with forward Euler, which I had written as:

```
def forward(dt)
  old_acc = []
  @body.each_index{|i| old_acc[i] = @body[i].acc}
  @body.each{|b| b.pos += b.vel*dt}
  @body.each_index{|i| @body[i].vel += old_acc[i]*dt}
end
```

The change is quite minimal: there is only the extra argument in `acc` and everything else remains the same:

```
def forward(dt)
  old_acc = []
  @body.each_index{|i| old_acc[i] = @body[i].acc(@body)}
  @body.each{|b| b.pos += b.vel*dt}
  @body.each_index{|i| @body[i].vel += old_acc[i]*dt}
end
```

But I must admit, I don't particularly like to make that one line longer, especially since it breaks the nice symmetry between `pos` and `vel` on the one hand, and `acc` on the other hand.

Alice: But that symmetry is only superficial, and in fact quite dangerous: `pos` and `vel` are actual variables whereas `acc` is a method. If you do not change `pos` explicitly, you can count on it to keep its old values. However, the same is not true for `acc`. You can call `acc` and then when you change `pos` and call `acc` again, you get a different value. In fact, as soon as you change the position `pos` for only one particle, a call to `acc` for *any* particle will be affected!

It is much better to warn yourself of this side effect, by making it clear that there is a hidden dependency: and the best way to show this dependency is by expressing it in the form of an argument to the method `acc`. In that case the dependency is no longer hidden, and the user is warned of the fact that `acc` depends on the states of all the bodies.

Bob: I see your point, but I still like the symmetry of my original notation. In any case, I agree that I should have commented the hidden dependency, at the very least in the form of an explicit comment. Meanwhile, let me make the same changes for any call to `acc` in any of the other integrators.

Alice: And don't forget to make the change in `epot` as well.

Bob: Ah yes, I had already forgotten about that one. In any case, the interpreter would have complained about a wrong number of arguments in `epot`, but let me modify `epot` as well.

5.8 Two Tests

Alice: Let's now do the same tests, for our generic two-body problem, and our figure-8 three-body problem.

Bob: Okay, here is the two-body problem that we started our testing cycle with, but now with the new code:

```
|gravity> ruby rknbody3a_driver.rb < test1.in
dt = 0.0001
dt_dia = 10
dt_out = 10
dt_end = 10
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0.02 , E_pot =  -0.16 , E_tot = -0.14
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 10, after 100000 steps :
  E_kin = 0.0887 , E_pot =  -0.229 , E_tot = -0.14
             E_tot - E_init = 4.55e-15
  (E_tot - E_init) / E_init = -3.25e-14
2
  9.9999999999900329e+00
  8.0000000000000004e-01
  1.1992351097726084e-01 -7.2126916688572407e-02
  2.0616138205436191e-01  4.2779060839347856e-02
  2.0000000000000001e-01
 -4.7969404390904336e-01  2.8850766675428963e-01
 -8.2464552821744763e-01 -1.7111624335739142e-01
```

Alice: Good! The same results as for the previous version of the code.

Bob: And here is the new figure-8 result:

```
|gravity> ruby rknbody3b_driver.rb < figure8.in
dt = 0.001
dt_dia = 2.1088
dt_out = 2.1088
dt_end = 2.1088
method = rk4
at time t = 0, after 0 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2.109, after 2109 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = -2e-15
  (E_tot - E_init) / E_init = 1.55e-15
3
  2.1089999999998787e+00
  1.0000000000000000e+00
 -1.6047303546488470e-04 -1.9320664965417420e-04
 -9.3227640249930266e-01 -8.6473492670753516e-01
  1.0000000000000000e+00
  9.7020367429337440e-01 -2.4296620300772800e-01
  4.6595057278750124e-01  4.3244644507801255e-01
  1.0000000000000000e+00
 -9.7004320125790211e-01  2.4315940965738195e-01
  4.6632582971180025e-01  4.3228848162952316e-01
```

Alice: And that's the same as well. It seems that you have correctly transformed the code from doubly-linked to singly-linked.

Chapter 6

Returning to Simplicity

6.1 Extra Body Variables

Bob: Now that we've begun to modify my original N-body code, I would like to write a third version, where I keep local copies of the auxiliary variables, such as `old_acc`, `half_vel`, and the like, as instance variables of the `Body` class. That should make the notation within the integrator methods a lot simpler. I don't like the notion of proliferating instance variables, though.

Alice: I don't think that would be such a bad thing. One could argue that in object-oriented programming, it is appropriate to modify the object if you use it for a different purpose. And especially in Ruby, such a modification will be easy. You can always add a few lines in a new class definition, and as you know, those new lines will be directly added to the existing definitions.

Bob: Yeah, well, I still prefer to be parsimonious. But I'm curious to see how much simpler the integrators will become. The first step will be to add those auxiliary variables to the `Body` class. In the first version of the code, I had:

```
attr_accessor :mass, :pos, :vel, :nb
```

In the alternative version, we left out the backward link:

```
attr_accessor :mass, :pos, :vel
```

And now let me put in the extra variables:

```
attr_accessor :mass, :pos, :vel, :old_pos, :half_vel, :a0, :a1, :a2
```

Ah, this is the only modification that we have to make to the `Body` class, so you're right, it is not as bad as I thought. All other changes happen only in the `Nbody` class!

6.2 Alternatives

Alice: But is this really necessary? I thought that in Ruby you can take any class definition, whether you defined the class yourself or whether you get it from a library, and extend that class by adding something like

```
class Body
  . . .
end
```

Bob: Yes, that is true. Indeed, we have used that procedure to define the method `to_v` for the `Array` class. However, in our code we have two class definitions directly following each other in a single file. The first one gives the `Body` class, and the second one gives the `Nbody` class. I suppose I could have given three definitions: first a standard one for `Body`, then the particular extension in the form of a second, additional definition for `Body` as you just indicated, and finally the `Nbody` class definition. But since everything is bundled in the same file anyway, that seemed to me to be a rather unnecessary maneuver.

Alice: Well, you know I like modularity. And you yourself were just complaining about the fact that a modification in the `Nbody` class forced you to intrude in the earlier definition of the `Body` class. At the very least, writing three definitions that way will make it more clear what is going on.

In fact, I would probably prefer to put the standard `Body` class definition in one file, and the `Nbody` class definition in another file. In that case, you can add the necessary `Body` extensions to the second file, together with the `Nbody` class definition that triggered the extensions.

Bob: I don't like to create a plethora of small files. But I see your point. And come to think of it, there must be other solutions.

Alice: Well, the best thing would be if you could modify the definition of the `Body` class from within `Nbody`, but I'm pretty sure that would violate some Ruby rules.

Bob: Actually, I'm not so sure. In Ruby you can do almost anything. The trick is to find out how. Hmmm. I have read something about a Ruby class called `Binding`, and a method `binding`, both of which seem to be related to creating the possibility to do just the sort of thing you brought up.

Alice: What do you know. Well, not now, I suggest.

Bob: I agree.

6.3 Forward

Alice: Let's start again with the forward Euler method, listing now all three versions in order: your original one; the one we got by including an explicit parameter in the `acc` call; and the new version you're writing now with extra variables in the `Body` class:

```
def forward(dt)
  old_acc = []
  @body.each_index{|i| old_acc[i] = @body[i].acc}
  @body.each{|b| b.pos += b.vel*dt}
  @body.each_index{|i| @body[i].vel += old_acc[i]*dt}
end
```

```
def forward(dt)
  old_acc = []
  @body.each_index{|i| old_acc[i] = @body[i].acc(@body)}
  @body.each{|b| b.pos += b.vel*dt}
  @body.each_index{|i| @body[i].vel += old_acc[i]*dt}
end
```

```
def forward(dt)
  @body.each{|b| b.old_acc = b.acc(@body)}
  @body.each{|b| b.pos += b.vel*dt}
  @body.each{|b| b.vel += b.old_acc*dt}
end
```

6.4 Clean Code

Bob: I must say, it is very gratifying to see how much cleaner this last version looks. Let me rewrite the other three integration methods as well, and call this file `rknbody4.rb`. Here they all are:

```
def leapfrog(dt)
  @body.each{|b| b.vel += b.acc(@body)*0.5*dt}
  @body.each{|b| b.pos += b.vel*dt}
  @body.each{|b| b.vel += b.acc(@body)*0.5*dt}
end
```

```
def rk2(dt)
  @body.each{|b| b.old_pos = b.pos}
  @body.each{|b| b.half_vel = b.vel + b.acc(@body)*0.5*dt}
  @body.each{|b| b.pos += b.vel*0.5*dt}
  @body.each{|b| b.vel += b.acc(@body)*dt}
  @body.each{|b| b.pos = b.old_pos + b.half_vel*dt}
end
```

```
def rk4(dt)
  @body.each{|b| b.old_pos = b.pos}
  @body.each{|b| b.a0 = b.acc(@body)}
  @body.each{|b| b.pos = b.old_pos + b.vel*0.5*dt + b.a0*0.125*dt*dt}
  @body.each{|b| b.a1 = b.acc(@body)}
  @body.each{|b| b.pos = b.old_pos + b.vel*dt + b.a1*0.5*dt*dt}
  @body.each{|b| b.a2 = b.acc(@body)}
  @body.each{|b| b.pos = b.old_pos + b.vel*dt + (b.a0+b.a1*2)*(1/6.0)*dt*dt}
  @body.each{|b| b.vel += (b.a0+b.a1*4+b.a2)*(1/6.0)*dt}
end
```

Alice: A lot easier to read. A great improvement over the previous two versions, though not *quite* as clean as the two-body version. Let's put up the `rk4` method for the old two-body code:

```
def rk4(dt)
  old_pos = pos
  a0 = acc
  @pos = old_pos + vel*0.5*dt + a0*0.125*dt*dt
  a1 = acc
  @pos = old_pos + vel*dt + a1*0.5*dt*dt
  a2 = acc
  @pos = old_pos + vel*dt + (a0+a1*2)*(1/6.0)*dt*dt
  @vel = vel + (a0+a1*4+a2)*(1/6.0)*dt
end
```

Bob: The difference is that in our latest version we still have to indicate which body `b` it is that gets the instructions, hence the "`b.`" in front of each `pos`, `vel`, *etc.*, call.

6.5 Sending a String

Alice: Well, I have an idea. Perhaps we can get rid of the "`b.`" in front of each physical variable, after all.

Bob: How? I mean, you have to loop over each particle! I can't see how you can get rid of that.

Alice: You can't get rid of that, I agree, but you don't have to repeat the fact that you are looping many times in one line, as we are doing now. I think we can express the idea of a loop just once in each line.

Bob: I still don't see how you can do that. In the old two-body code, we could get away with writing `pos += vel*dt` because there was only one `pos` and one `vel`, but here we have little choice but writing `b.pos += b.vel*dt`, at a minimum. And I already chose the shortest name I could think of, `b` for the body whose `pos` gets updated!

Alice: My idea is to let the `Nbody` class give a command to the `Body` class, specifying directly to do `pos += vel*dt` for particle `b`, without repeating the `b` presence separately for `pos` and `vel`.

Bob: That would be nice, yes, but it still looks impossible to do, since it would involve something sending that whole line to the `Body` class!

Alice: Exactly!

Bob: I beg your pardon?

Alice: You got it! Let us send that line to the `Body` class! We can ask the integration methods in `Nbody` to specify what needs to be done, but instead of making the actual command calls, these methods can write the commands into a string, and then pass that string down to the `Body` class.

Bob: Hey, that is a great idea! I would never have thought about that. Can you really do that? Well, of course you can. I guess I'm still thinking too much in Fortran and C terms. But I must say, I'm beginning to get really worried about speed; this may slow down execution of the code even more. And Ruby is already slow to begin with.

Alice: Let us postpone efficiency discussions for later. I'm pretty hopeful that we can find ways to speed things up later. In the worst case we can switch back to a more traditional form in our final production code. If we can make the prototyping process easier and more transparent, that would already be a big gain. I care less about elegance of the final version than I do about the intermediate test versions.

Bob: Well, okay, but we shouldn't wait *too* long in checking the speed. I would feel more comfortable if we had some hard-nosed proof that a Ruby-based N-body code can really compete with codes written in C or Fortran. Meanwhile, I must admit, I do like the flexibility of Ruby. What a trick, to implement message passing between the Nbody system scheduler and the individual bodies, literally by passing messages in the form of a string!

Alice: Yes, Ruby invites a different way of thinking. Even if you could find a way to do such a trick in C++, in some convolved way, the point is that it would not occur to you to do so, since the language does not invite that way of thinking.

Bob: Quite convolved: for one thing, you would need to find a way to implement a form of dynamic loading! C++ is a compiled language, not an interpreted language like Ruby.

Alice: Ah, yes, of course, I'm already beginning to forget what it means to work with compile cycles in prototyping code!

6.6 Wishful Thinking

Bob: How did you get this idea, of using message passing?

Alice: When I browsed through some Ruby code on the web, I came across something similar as what I just suggested, and then I realized that that would be a very natural way of passing instructions between classes.

Bob: Let's try it! I'll create yet another file, `rknbody5.rb`, and see how we can implement your idea.

Alice: This time, I suggest we start on the level of the `Nbody` class. In a wishful thinking sort of way, let us assume that we can tell a `Body` named `b` to calculate something, by issuing a command `b.calc(s)`, where `s` is a string that will get executed by `b`. The forward Euler method would then look like this:

```
def forward(dt)
  @body.each{|b| b.calc(@body,dt," @old_acc = acc(ba) ")}
  @body.each{|b| b.calc(@body,dt," @pos += @vel*dt ")}
  @body.each{|b| b.calc(@body,dt," @vel += @old_acc*dt ")}
end
```

Bob: And indeed, with no mention of `b` anymore within the string that is passed as the third argument of `calc`.

Alice: The first two arguments are needed, because otherwise the `Body` class will not know what to do with the string: it knows about its own instance variables such as `@pos`, and it knows about the *method* `acc`, but not about the *argument* to `acc`, which has to be specified explicitly.

Bob: I see. What does `ba` stand for?

Alice: Body array. When `calc` executes the call, it will replace `ba` by its first argument, `@body`. Similarly, it will replace `dt` by `dt`. I'll show you in a moment. I hope it will all work. And I'm pretty sure it will.

6.7 Implementation

Bob: You used the `calc` method in the middle line of `forward` as well, even though you could have used the simpler statement

```
@body.each{|b| b.pos += b.vel*dt}
```

which we used before, in the previous version.

Alice: Yes, but my intention was to not break the symmetry between the lines. By treating all of them in the same way, your eye can be guided to what is different on each line, forgetting the left half of each line, which is the same in all cases. Here, let me write the other three methods as well, and then it will become more clear how the actual strings that contain the commands will stand out:

```
def leapfrog(dt)
  @body.each{|b| b.calc(@body,dt," @vel += acc(ba)*0.5*dt ")}
  @body.each{|b| b.calc(@body,dt," @pos += @vel*dt ")}
  @body.each{|b| b.calc(@body,dt," @vel += acc(ba)*0.5*dt ")}
end
```

```
def rk2(dt)
  @body.each{|b| b.calc(@body,dt," @old_pos = @pos ")}
  @body.each{|b| b.calc(@body,dt," @half_vel = @vel + acc(ba)*0.5*dt ")}
  @body.each{|b| b.calc(@body,dt," @pos += @vel*0.5*dt ")}
  @body.each{|b| b.calc(@body,dt," @vel += acc(ba)*dt ")}
  @body.each{|b| b.calc(@body,dt," @pos = @old_pos + @half_vel*dt ")}
end
```

```
def rk4(dt)
  @body.each{|b| b.calc(@body,dt," @old_pos = @pos ")}
  @body.each{|b| b.calc(@body,dt," @a0 = acc(ba) ")}
  @body.each{|b| b.calc(@body,dt," @pos = @old_pos +
                                    @vel*0.5*dt + @a0*0.125*dt*dt ")}
  @body.each{|b| b.calc(@body,dt," @a1 = acc(ba) ")}
  @body.each{|b| b.calc(@body,dt," @pos = @old_pos +
                                    @vel*dt + @a1*0.5*dt*dt ")}
  @body.each{|b| b.calc(@body,dt," @a2 = acc(ba) ")}
  @body.each{|b| b.calc(@body,dt," @pos = @old_pos +
                                @vel*dt + (@a0+@a1*2)*(1/6.0)*dt*dt ")}
  @body.each{|b| b.calc(@body,dt," @vel += (@a0+@a1*4+@a2)*(1/6.0)*dt ")}
end
```

Bob: It's an improvement over the first two versions, which used an index i. In the previous version, we could leave out that obnoxious i index at the expense of introducing extra variables on the Body level. But now we can have our cake and eat it: no more i and no more extra variables with Body, I presume.

Alice: Correct! The content of the string effectively will declare the extra variables for us when the string is evaluated in the Body class. Here is how I would write the calc method for the Body class:

```
def calc(body_array, time_step, s)
  ba  = body_array
  dt = time_step
  eval(s)
end
```

Bob: Simplicity itself. I see now what you meant, when you described the way the two parameters ba and dt were going to be substituted in an actual call to calc.

Alice: Almost too simple to be true, but I think this is all correct.

6.8 Indirect String Sending

Bob: Shall we move on? Or do you have a suggestion for further improvements?

Alice: I must say that the integration methods still look too cluttered for my taste. They miss the simple elegance and brevity of expression of our previous version. For one thing, in that version we did not have to break any statement up over two lines. What bothers me especially is that for most statements, more than half of the line gets repeated exactly.

Bob: I wonder whether we can do something about that.

Alice: I think we can. So far, we have introduced a calc function on the Body level. How about introducing a second calc function on the Nbody level? Let's create one more file, rknbody6.rb, in which we give the Nbody class the following extra method:

```
def calc(y,s)
  @body.each{|b| b.calc(@body,y,s)}
end
```

Bob: Brevity indeed. I see what you mean. Forward Euler then becomes, instead of

```
  def forward(dt)
    @body.each{|b| b.calc(@body,dt," @old_acc = acc(ba) ")}
    @body.each{|b| b.calc(@body,dt," @pos += @vel*dt ")}
    @body.each{|b| b.calc(@body,dt," @vel += @old_acc*dt ")}
  end
```

which we just wrote, quite a bit shorter as:

```
  def forward(dt)
    calc(dt," @old_acc = acc(ba) ")
    calc(dt," @pos += @vel*dt ")
    calc(dt," @vel += @old_acc*dt ")
  end
```

Alice: Exactly. And the following three methods become:

```
  def leapfrog(dt)
    calc(dt," @vel += acc(ba)*0.5*dt ")
    calc(dt," @pos += @vel*dt ")
    calc(dt," @vel += acc(ba)*0.5*dt ")
  end
```

```
  def rk2(dt)
    calc(dt," @old_pos = @pos ")
    calc(dt," @half_vel = @vel + acc(ba)*0.5*dt ")
    calc(dt," @pos += @vel*0.5*dt ")
    calc(dt," @vel += acc(ba)*dt ")
    calc(dt," @pos = @old_pos + @half_vel*dt ")
  end
```

```
  def rk4(dt)
    calc(dt," @old_pos = @pos ")
    calc(dt," @a0 = acc(ba) ")
    calc(dt," @pos = @old_pos + @vel*0.5*dt + @a0*0.125*dt*dt ")
    calc(dt," @a1 = acc(ba) ")
    calc(dt," @pos = @old_pos + @vel*dt + @a1*0.5*dt*dt ")
    calc(dt," @a2 = acc(ba) ")
    calc(dt," @pos = @old_pos + @vel*dt + (@a0+@a1*2)*(1/6.0)*dt*dt ")
    calc(dt," @vel += (@a0+@a1*4+@a2)*(1/6.0)*dt ")
  end
```

6.9 The Same, Yet Different

Bob: I like that: every statement now fits on one line, we don't have to modify or extend the `Body` class when we introduce a new integration scheme, and we don't mention the variable `@body` anymore in each line. I'm satisfied: this combines all good things in one. And I must say, I'm growing fond of working with Ruby.

Alice: So do I. Even though I knew that Ruby was a well designed language, I was a bit skeptical at first about how much that would really buy us. But as we already have seen, it buys us quite a bit in terms of clarity of expression. And when writing complicated programs and packages, something we will start doing soon, clarity of expression is more important than anything else. Nothing else will allow you to maintain an overview over the whole situation.

However, just one point: you mentioned that the `Body` class did not need to be extended. But that is not true. We *have* extended the `Body` class by adding a `calc` method to i.

Bob: I meant that we don't have to make a different and separate extension to the `Body` class, each time we introduce a new integrator in `Nbody`. We only modify `Body` once, by adding `calc`, and the same `calc` will do different things for different integrators.

Alice: Which means that in practice, dynamically, we *are* augmenting the `Body` class, each time we add an integrator to the `Nbody` class.

Bob: That is true, but it is invisible as far as the code is concern. If the `Body` class would be written in a separate file, that file would not have to be changed, upon adding an integrator.

Alice: But I thought you didn't like to cut up our file into, what did you call it again, ah yes, a *plethora* of small files.

Bob: Very funny. I'm not really suggesting to split up the file, I just tried to make a clear point even clearer. But of course we are both right: I am right to say that the *written definition* of the `Body` class remains unchanged, and you are right if you say that the *actual definition*, as it is dynamically changed during execution of the code by the interpreted, *does* change.

Alice: Right you are! And right I am. Okay, let's move on.

6.10 Testing

Bob: With all this bickering, we haven't tested yet any of our latest three versions. For now, let's just try the figure-8 triple. I'll run the first singly-linked code version again, to make sure we got the right output.

```
|gravity> ruby rknbody3b_driver.rb < figure8.in
```

```
dt = 0.001
dt_dia = 2.1088
dt_out = 2.1088
dt_end = 2.1088
method = rk4
at time t = 0, after 0 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2.109, after 2109 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = -2e-15
  (E_tot - E_init) / E_init = 1.55e-15
3
  2.1089999999998787e+00
  1.0000000000000000e+00
 -1.6047303546488470e-04 -1.9320664965417420e-04
 -9.3227640249930266e-01 -8.6473492670753516e-01
  1.0000000000000000e+00
  9.7020367429337440e-01 -2.4296620300772800e-01
  4.6595057278750124e-01  4.3244644507801255e-01
  1.0000000000000000e+00
 -9.7004320125790211e-01  2.4315940965738195e-01
  4.6632582971180025e-01  4.3228848162952316e-01
```

And then our version with the extra variables added by hand to the `Body` class:

```
|gravity> ruby rknbody4a_driver.rb < figure8.in
dt = 0.001
dt_dia = 2.1088
dt_out = 2.1088
dt_end = 2.1088
method = rk4
at time t = 0, after 0 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2.109, after 2109 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = -2e-15
  (E_tot - E_init) / E_init = 1.55e-15
3
  2.1089999999998787e+00
  1.0000000000000000e+00
 -1.6047303546488470e-04 -1.9320664965417420e-04
```

```
 -9.3227640249930266e-01 -8.6473492670753516e-01
  1.0000000000000000e+00
  9.7020367429337440e-01 -2.4296620300772800e-01
  4.6595057278750124e-01  4.3244644507801255e-01
  1.0000000000000000e+00
 -9.7004320125790211e-01  2.4315940965738195e-01
  4.6632582971180025e-01  4.3228848162952316e-01
```

Good! Now the version that is sending a string from `Nbody` to `Body`:

```
|gravity> ruby rknbody5a_driver.rb < figure8.in
dt = 0.001
dt_dia = 2.1088
dt_out = 2.1088
dt_end = 2.1088
method = rk4
at time t = 0, after 0 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2.109, after 2109 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = -2e-15
  (E_tot - E_init) / E_init = 1.55e-15
3
  2.1089999999998787e+00
  1.0000000000000000e+00
 -1.6047303546488470e-04 -1.9320664965417420e-04
 -9.3227640249930266e-01 -8.6473492670753516e-01
  1.0000000000000000e+00
  9.7020367429337440e-01 -2.4296620300772800e-01
  4.6595057278750124e-01  4.3244644507801255e-01
  1.0000000000000000e+00
 -9.7004320125790211e-01  2.4315940965738195e-01
  4.6632582971180025e-01  4.3228848162952316e-01
```

Also perfect. Finally our last version with the two `calc` methods:

```
|gravity> ruby rknbody6a_driver.rb < figure8.in
dt = 0.001
dt_dia = 2.1088
dt_out = 2.1088
dt_end = 2.1088
```

```
method = rk4
at time t = 0, after 0 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2.109, after 2109 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = -2e-15
  (E_tot - E_init) / E_init = 1.55e-15
3
  2.1089999999998787e+00
  1.0000000000000000e+00
 -1.6047303546488470e-04 -1.9320664965417420e-04
 -9.3227640249930266e-01 -8.6473492670753516e-01
  1.0000000000000000e+00
  9.7020367429337440e-01 -2.4296620300772800e-01
  4.6595057278750124e-01  4.3244644507801255e-01
  1.0000000000000000e+00
 -9.7004320125790211e-01  2.4315940965738195e-01
  4.6632582971180025e-01  4.3228848162952316e-01
```

Great! It all works.

Alice: That's very nice. As we already said, almost too good to believe.

Chapter 7

A Final Version

7.1 Clarity

Bob: It was fun to play with so many different versions, but I'm beginning to get a little confused as to which version did what. Maybe we should pick just one version, and use that while we add more features and move toward specific applications.

Alice: I agree. And since we decided not to worry, for now at least, about performance, let us concentrate on clarity of expression. I must say, I like the last one best, in file `rknbody6.rb`, where everything fits on one line. However, the version where we gave the `Body` class explicit helper variables, in file `rknbody4.rb`, was even shorter.

Bob: Yeah, I did not particularly like the idea of giving this poor `Body` class all possible helper variables for all possible application. Even though we discussed more clever ways to do this, frankly, I don't care too much what we will choose in the end. I liked your suggestion to send a command string to be evaluated dynamical, thereby generating the proper helper variables, mostly because of its novelty.

Alice: And it goes with the spirit of the times: just-in-time-delivery! But what I like best about this latest method is that it obeys the DRY principle: we are not repeating ourselves.

Bob: Apart from the fact that you repeatedly bring up particular principles.

Alice: I'll repeatedly ignore that. Apart from that point, when clarity is really the criterion, I am not sure whether the last version is really clearer. Let us compare the forward Euler algorithm in both cases, in `rknbody4.rb`:

```
  def forward(dt)
    @body.each{|b| b.old_acc = b.acc(@body)}
```

```
    @body.each{|b| b.pos += b.vel*dt}
    @body.each{|b| b.vel += b.old_acc*dt}
  end
```

and in `rknbody6.rb`

```
  def forward(dt)
    calc(dt," @old_acc = acc(ba) ")
    calc(dt," @pos += @vel*dt ")
    calc(dt," @vel += @old_acc*dt ")
  end
```

Bob: The last one is clearly shorter.

Alice: That I don't mind so much. I'm just not happy with the fact that it is not clear, for a casual reader, what that variable `dt` is doing there, as the first two arguments of `calc`, and the appearance of `ba` is also a mystery; there is no indication here that `ba` stands for 'body array' and will get its value from `@body`, somewhere else. In contrast, the earlier version has nothing hidden: the `@body.each{|b| . . . }` construct is vanilla flavor for someone familiar with Ruby.

7.2 Brevity

Bob: Perhaps we can improve the `calc` method of `Nbody` further. How about redefining it in such a way that we can leave out the first argument altogether?

Alice: Ah, that's a good idea. It is also a logical next step, after introducing the shortcut notion of sending a string in the first place. Once we do something that is somewhat dirty and not so self-explanatory, we might as well go all the way.

Bob: I suppose that we would have to introduce an extra instance variable `@dt` for `Nbody`. Otherwise it will not be possible to remove the first argument `dt` from the current `calc`. In fact, that would make the definition even shorter. So it would look very clean, like this:

```
  def forward
    calc(" @old_acc = acc(ba) ")
    calc(" @pos += @vel*dt ")
    calc(" @vel += @old_acc*dt ")
  end
```

Note that I have created yet another version of our code, in file `rknbody7.rb`.

Alice: That is short and sweet, indeed. And we have to modify `calc` on the `Nbody` level from what we had before:

```
def calc(y,s)
  @body.each{|b| b.calc(@body,y,s)}
end
```

to a version with only one parameter, namely the command string:

```
def calc(s)
  @body.each{|b| b.calc(@body, @dt, s)}
end
```

Since the other two parameters to the `calc` method of `Body` are now both instance variables, their value is known here.

Bob: What else do we have to change? We have to set the value of the new variable `@dt` in the `evolve` method, and we have to leave out the `dt` argument when invoking the integration methods. Two small changes, which leaves us with `evolve` looking like this:

```
def evolve(integration_method, dt, dt_dia, dt_out, dt_end)
  nsteps = 0
  e_init
  write_diagnostics(nsteps)
  @dt = dt
  t_dia = dt_dia - 0.5*@dt
  t_out = dt_out - 0.5*@dt
  t_end = dt_end - 0.5*@dt

  while @time < t_end
    send(integration_method)
    @time += @dt
    nsteps += 1
    if @time >= t_dia
      write_diagnostics(nsteps)
      t_dia += dt_dia
    end
    if @time >= t_out
      simple_print
```

```
      t_out += dt_out
    end
  end
end
```

7.3 Correctness

Alice: And the only other changes, besides the change in the forward Euler algorithm we already saw, are the simplified readings of the three other integrators. They now become:

```
def leapfrog
  calc(" @vel += acc(ba)*0.5*dt ")
  calc(" @pos += @vel*dt ")
  calc(" @vel += acc(ba)*0.5*dt ")
end
```

```
def rk2
  calc(" @old_pos = @pos ")
  calc(" @half_vel = @vel + acc(ba)*0.5*dt ")
  calc(" @pos += @vel*0.5*dt ")
  calc(" @vel += acc(ba)*dt ")
  calc(" @pos = @old_pos + @half_vel*dt ")
end
```

```
def rk4
  calc(" @old_pos = @pos ")
  calc(" @a0 = acc(ba) ")
  calc(" @pos = @old_pos + @vel*0.5*dt + @a0*0.125*dt*dt ")
  calc(" @a1 = acc(ba) ")
  calc(" @pos = @old_pos + @vel*dt + @a1*0.5*dt*dt ")
  calc(" @a2 = acc(ba) ")
  calc(" @pos = @old_pos + @vel*dt + (@a0+@a1*2)*(1/6.0)*dt*dt ")
  calc(" @vel += (@a0+@a1*4+@a2)*(1/6.0)*dt ")
end
```

Bob: Time to check whether the new code does the same thing as all the older ones:

```
|gravity> ruby rknbody7a_driver.rb < figure8.in
dt = 0.001
dt_dia = 2.1088
dt_out = 2.1088
dt_end = 2.1088
method = rk4
at time t = 0, after 0 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2.109, after 2109 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = -2e-15
  (E_tot - E_init) / E_init = 1.55e-15
3
  2.1089999999998787e+00
  1.0000000000000000e+00
 -1.6047303546488470e-04 -1.9320664965417420e-04
 -9.3227640249930266e-01 -8.6473492670753516e-01
  1.0000000000000000e+00
  9.7020367429337440e-01 -2.4296620300772800e-01
  4.6595057278750124e-01  4.3244644507801255e-01
  1.0000000000000000e+00
 -9.7004320125790211e-01  2.4315940965738195e-01
  4.6632582971180025e-01  4.3228848162952316e-01
```

Alice: And so it does. Good! I think we can stick with this version as our work horse, for a while at least.

7.4 More Information

Bob: There is one improvement I would like to add. Remember when we were debugging our code and we were analyzing a single time step, trying to figure out whether the position and velocity were updated correctly?

Alice: Yes, that was a simple situation, in the case of the forward Euler algorithm, and we could analytically calculate the acceleration.

Bob: Of course, in general we can't be so lucky, and I would prefer to have an easy way to ask for an output of the acceleration as well. Ah, come to think of it, we have this pretty print output version, that we wrote long ago, but that never used since. Here it is:

```
  def pp                                # pretty print
    print "     N = ", @body.size, "\n"
```

```
    print "  time = ", @time, "\n"
    @body.each{|b| b.pp}
  end
```

All it does is invoke the pretty print version for each particle:

```
  def pp               # pretty print
    print to_s
  end
```

which does nothing else but print the `Body` instance in its standard string form:

```
  def to_s
    "  mass = " + @mass.to_s + "\n" +
    "   pos = " + @pos.join(", ") + "\n" +
    "   vel = " + @vel.join(", ") + "\n"
  end
```

Well, let us just add an extra line for the acceleration. Since pretty print became `pp`, a pretty print with extras should be called a `ppx`:

```
  def ppx(body_array)             # pretty print, with extra information (acc)
    STDERR.print to_s + "   acc = " + acc(body_array).join(", ") + "\n"
  end
```

I will put this version in file `rknbody8.rb`.

Alice: That makes a lot of sense. You choose the standard error output stream `STDERR` because you don't want this extra debugging information to be mixed up with the particle output snapshot, which is written on the default standard output stream, and which can be piped into another program.

And of course, you have to give `ppx` a parameter, namely the array of all the particles in the N-body system, otherwise our particle cannot compute the acceleration that it receives from all other particles. Let's see. This means that `ppx` on the `Body` level must be invoked from the `Nbody` as:

```
  def ppx                         # pretty print, with extra information (acc)
    print "     N = ", @body.size, "\n"
    print "  time = ", @time, "\n"
    @body.each{|b| b.ppx(@body)}
  end
```

7.5 An Initial Snapshot Output

Bob: Indeed. Now what else is there left to do? We don't want to have this extra information all the time, since it would clutter up the output. Let us introduce a special flag, `x_flag`, a boolean variable that will be set to 'true' if the extra output is desired, and set to 'false' when we don't need it.

This means that the method that writes the diagnostics will now get `x_flag` as a second parameter, and a few extra lines at the end, to invoke `ppx` if the flag is set to be 'true':

```
  def write_diagnostics(nsteps, x_flag)
    etot = ekin + epot
    STDERR.print <<END

at time t = #{sprintf("%g", time)}, after #{nsteps} steps :
  E_kin = #{sprintf("%.3g", ekin)} ,\
 E_pot =  #{sprintf("%.3g", epot)} ,\
 E_tot = #{sprintf("%.3g", etot)}
            E_tot - E_init = #{sprintf("%.3g", etot - @e0)}
  (E_tot - E_init) / E_init = #{sprintf("%.3g", (etot - @e0)/@e0 )}
END
    if x_flag
      STDERR.print "  for debugging purposes, here is the internal data ",
                   "representation:\n"
      ppx
    end
  end
```

Alice: Now that we are adding features, I would like to have the option to echo the initial snapshot, the data file that is read in before any integration step is taken. I know that I can always read that information myself from the input file, but sometimes it is nice to have it right in front of you, together with the other data,

Bob: Yes, especially when are debugging and you want to take a single time step. Okay, let us introduce a second flag `init_out`. If the value of this flag is 'true', we will require an initial output, in the form of a snapshot on the standard output stream. If the value is 'false', we skip the initial output.

We can implement the effects of both flags, `x_flag` and `init_out`, by passing them as additional parameters to `evolve`. The modifications to the body of this method are then very minor. We only have to change three lines. First, the two calls to `write_diagnostics` acquire `x_flag` as extra parameter, as we have already seen. Second, we add an extra line

```
    simple_print if init_out
```

just before we start the `while` loop. The new version of the `evolve` method thus becomes:

```
def evolve(integration_method, dt, dt_dia, dt_out, dt_end, init_out, x_flag)
  nsteps = 0
  e_init
  write_diagnostics(nsteps, x_flag)
  @dt = dt
  t_dia = dt_dia - 0.5*dt
  t_out = dt_out - 0.5*dt
  t_end = dt_end - 0.5*dt

  simple_print if init_out

  while @time < t_end
    send(integration_method)
    @time += dt
    nsteps += 1
    if @time >= t_dia
      write_diagnostics(nsteps, x_flag)
      t_dia += dt_dia
    end
    if @time >= t_out
      simple_print
      t_out += dt_out
    end
  end
end
```

Alice: An alternative would have been to make both flags into instance variables for the `Nbody` class.

Bob: Yes, that would perhaps look more tidy, giving us fewer arguments to pass around. Either way, I don't care very much: most class definitions involve a trade off between the number of instance variables and the number of arguments being passed around. I'm happy just to keep them as arguments for now.

7.6 A New Driver

Alice: Let's make the necessary changes in the driver file as well:

```
require "rknbody8.rb"
```

```
include Math

dt = 0.001           # time step
dt_dia = 2.1088      # diagnostics printing interval
dt_out = 2.1088      # output interval
dt_end = 2.1088      # duration of the integration
init_out = false     # initial output requested ?
x_flag = false       # extra diagnostics requested ?
##method = "forward"  # integration method
##method = "leapfrog" # integration method
##method = "rk2"      # integration method
method = "rk4"       # integration method

STDERR.print "dt = ", dt, "\n",
      "dt_dia = ", dt_dia, "\n",
      "dt_out = ", dt_out, "\n",
      "dt_end = ", dt_end, "\n",
      "init_out = ", init_out, "\n",
      "x_flag = ", x_flag, "\n",
      "method = ", method, "\n"

nb = Nbody.new
nb.simple_read
nb.evolve(method, dt, dt_dia, dt_out, dt_end, init_out, x_flag)
```

7.7 A Final Test

Bob: All very straightforward indeed: at the end the two extra parameters for `evolve`, and above the introduction of the two flags in analogy with the other parameters. Let's test it out, to see whether we still get the same results for our figure out orbit:

```
|gravity> ruby rknbody8a_driver.rb < figure8.in
dt = 0.001
dt_dia = 2.1088
dt_out = 2.1088
dt_end = 2.1088
init_out = false
x_flag = false
method = rk4
at time t = 0, after 0 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = 0
```

```
  (E_tot - E_init) / E_init = -0
at time t = 2.109, after 2109 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = -2e-15
  (E_tot - E_init) / E_init = 1.55e-15
3
  2.1089999999998787e+00
  1.0000000000000000e+00
 -1.6047303546488470e-04 -1.9320664965417420e-04
 -9.3227640249930266e-01 -8.6473492670753516e-01
  1.0000000000000000e+00
  9.7020367429337440e-01 -2.4296620300772800e-01
  4.6595057278750124e-01  4.3244644507801255e-01
  1.0000000000000000e+00
 -9.7004320125790211e-01  2.4315940965738195e-01
  4.6632582971180025e-01  4.3228848162952316e-01
```

Alice: All is well, clearly. Good! So now we have a version of the code that is both lean and easy to read on the level of the integrators, and has extra debugging options. Progress!

Chapter 8

An Eight-Body System

8.1 Setting Up a Cube

Bob: Now that we have settled on a tool for doing N-body simulations, it would be a pity to stop with three bodies. Let's try it out on a bunch more particles.

Alice: We haven't yet written tools for setting up initial conditions, though, such as a Plummer model, or a King model, or even just a homogeneous sphere with particles sprinkled in. We will certainly do that later, but starting with that right know would be too much of a distraction. After all, it was you who wanted to move on quickly to graphics!

Bob: I agree. Let's do something really simple then. How about setting up eight particles on the eight corners of a cube, centered on the origin? We can start with all particles at rest, and just let them fall toward each other.

Alice: That sounds like a reasonably quick try. But we cannot give them equal masses, otherwise by symmetry they will all hit each other in the center, at distance zero from each other, where the inter-particle forces will be infinitely large.

Bob: Yes, we have to perturb something. Either we can give them equal masses, and small but different initial velocities, or we can give them zero velocities but slightly different masses. Let me do the latter. Here are some initial conditions:

```
8
0
1.0
1 1 1
0 0 0
1.1
1 1 -1
```

```
0 0 0
1.2
1 -1 1
0 0 0
1.3
1 -1 -1
0 0 0
1.4
-1 1 1
0 0 0
1.5
-1 1 -1
0 0 0
1.6
-1 -1 1
0 0 0
1.7
-1 -1 -1
0 0 0
```

As you can see, I took masses starting from 1.0 with increments of 0.1 for each next particle as I was walking around the eight corners of the central cube, for which the edges all have a length of 2.

Alice: The advantage of perturbing the masses, rather than the velocities, is that you keep the center of mass at rest. In other words, the kinetic energy you will be measuring, as soon as the particles start moving, will be the energy of the internal motion only. It will not receive a contribution from the kinetic energy associated with center-of-mass motion. If you had perturbed the velocities arbitrarily, that would no longer be true.

8.2 Letting Go

Bob: Let's guess how long would it take for the particles to reach the center. The masses are of order unity, the distances also, so the accelerations must also be of order unity. This would suggest that it would take of order one time unit for the particles to meet each other. Well, let us ask the computer to tell us whether it will take them more than one time unit or less.

Alice: Hmm, we should be able to predict that before doing a run. Wasn't it John Wheeler, who told us never to do a calculation before you know the answer? I like his attitude. Relying too much on raw computer power can make you lazy.

Bob: Lazy is in the eye of the beholder, I guess: it is a lot of work to write a good computer program, as we both know! But I see your point. It certainly doesn't hurt to try to predict numerical results beforehand, and it makes you more likely to catch a bug, if things come out differently from what you expected.

Alice: Not only that, it will give you more physical insight into the answer as well.

Bob: Okay, let's see whether we can predict the outcome of our particle race toward the center. I started saying that inter-particle distances were of order unity. However, the typical distances between particles are actually more like 2, 3 or 3.5, roughly speaking as an approximation for $2, 2\sqrt(2), 2\sqrt(3)$, depending whether they share an edge, a side, or nothing at all. So the initial acceleration, with inverse square forces, will receive contributions that have a distance dependence of something like $7(1/3)^2$, if we take 3 as a typical distance. With a typical mass being somewhat larger than 1, we do indeed get an acceleration that is fairly close to 1, between 0.75 and 1.5, I would guess.

Now this means that when you start from rest, and you have a distance to the center of $\sqrt(3)$ or roughly 1.7, it will take more than one time unit to arrive at the center. Of course, nonlinear effects will complicate things, but I don't think they will invalidate this simple reasoning so quickly. I'm pretty sure that by time `t = 1`, the particles haven't arrived in the center yet.

Alice: I agree. Okay, we have placed our bets. Let the truth be revealed! And actually, this would be a good time to use the `x_flag` that we built in to ask for extra information about accelerations.

Bob: Good idea. Okay, here goes, let's run things for one time unit:

```
|gravity> ruby rknbody8b_driver.rb < cube1.in
dt = 0.1
dt_dia = 1
dt_out = 10
dt_end = 1
init_out = false
x_flag = true
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0 , E_pot =  -20.7 , E_tot = -20.7
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
  for debugging purposes, here is the internal data representation:
  mass = 1.0
   pos = 1.0, 1.0, 1.0
   vel = 0.0, 0.0, 0.0
   acc = -0.705795165844984, -0.63811749631532, -0.604278661550489
  mass = 1.1
   pos = 1.0, 1.0, -1.0
   vel = 0.0, 0.0, 0.0
```

```
    acc = -0.725983913601737, -0.658306244072073, 0.556789739777578
   mass = 1.2
    pos = 1.0, -1.0, 1.0
    vel = 0.0, 0.0, 0.0
    acc = -0.74617266135849, 0.543139652769499, -0.644656157063995
   mass = 1.3
    pos = 1.0, -1.0, -1.0
    vel = 0.0, 0.0, 0.0
    acc = -0.766361409115244, 0.563328400526252, 0.597167235291084
   mass = 1.4
    pos = -1.0, 1.0, 1.0
    vel = 0.0, 0.0, 0.0
    acc = 0.515839478753342, -0.718872487342333, -0.685033652577501
   mass = 1.5
    pos = -1.0, 1.0, -1.0
    vel = 0.0, 0.0, 0.0
    acc = 0.536028226510095, -0.739061235099086, 0.637544730804591
   mass = 1.6
    pos = -1.0, -1.0, 1.0
    vel = 0.0, 0.0, 0.0
    acc = 0.556216974266848, 0.623894643796512, -0.725411148091007
   mass = 1.7
    pos = -1.0, -1.0, -1.0
    vel = 0.0, 0.0, 0.0
    acc = 0.576405722023601, 0.644083391553265, 0.677922226318097
at time t = 1, after 10 steps :
  E_kin = 12.3 , E_pot =  -33 , E_tot = -20.7
             E_tot - E_init = 0.000633
  (E_tot - E_init) / E_init = -3.05e-05
  for debugging purposes, here is the internal data representation:
  mass = 1.0
   pos = 0.597644451816504, 0.63640287626745, 0.655762419276459
   vel = -0.940991555344834, -0.850014228000948, -0.80458017381832
   acc = -1.71506911283474, -1.54866562852442, -1.4652605274951
  mass = 1.1
   pos = 0.585134269661813, 0.624094692341221, -0.682431788577738
   vel = -0.973577364186811, -0.881473975864098, 0.743630411318128
   acc = -1.79677056984513, -1.62409268319797, 1.36430528840404
  mass = 1.2
   pos = 0.572566955655305, -0.689897591000862, 0.631297823877706
   vel = -1.00651894751619, 0.727233325183985, -0.866662302150037
   acc = -1.88087240679652, 1.34171513605627, -1.6115608160994
  mass = 1.3
   pos = 0.559940111130521, -0.677904296108633, -0.658282111563603
   vel = -1.03983780651743, 0.756905613895535, 0.803924789872002
   acc = -1.96757353300315, 1.40685742201327, 1.50031839892619
```

```
mass = 1.4
 pos = -0.705282791726003, 0.586860755993474, 0.60663858079331
 vel = 0.69165870948931, -0.977682574453683, -0.92985560640001
 acc = 1.27847519172173, -1.86192287793191, -1.76451307558405
mass = 1.5
 pos = -0.693423631896722, 0.574338867566358, -0.633959293898993
 vel = 0.720463619202033, -1.01042621662886, 0.86512158769774
 acc = 1.33750716238934, -1.94561109813966, 1.64085701293565
mass = 1.6
 pos = -0.681529221962402, -0.64167993353146, 0.581767543659952
 vel = 0.749410199081073, 0.847120045366664, -0.99430431139073
 acc = 1.39673308307382, 1.60737124811481, -1.92531904373592
mass = 1.7
 pos = -0.669597957823065, -0.629517535170942, -0.609447786811517
 vel = 0.778508588720364, 0.877635982210738, 0.927342131031408
 acc = 1.45618251518339, 1.67618202793406, 1.78678060432799
   N = 8
time = 0.0
   N = 8
time = 1.0
```

8.3 Passing Through

Alice: And indeed, the particles did not reach the center yet. The first particle, for example, that started at the right-far-upper corner, at $\{x, y, z\} = \{1, 1, 1\}$ still has positive values for all three position components, and velocity components that are all negative, indicating that it is moving toward the center, but hasn't quite gotten there yet. It is about half way, judging from the size of the position components.

Bob: And look, my guestimate for the accelerations was correct too. Going back to the first snapshot output, typical components for the acceleration at time zero are 0.6 and 0.7, which means in three dimensions that the magnitude of the acceleration vector must be something like $0.65\sqrt(3)$, say, or about 1.1; comfortable within my predicted range!

Alice: Yes, well done! And our integrator has behaved well, too, even with the rather large time step of 0.1 that we have given it. Perhaps this is not surprising, given that the particles haven't reached the central crunch yet.

Bob: I bet things won't go so well for the next time unit. But let's try and see what happens:

```
|gravity> ruby rknbody8c_driver.rb < cube1.in
dt = 0.1
```

```
dt_dia = 2
dt_out = 10
dt_end = 2
init_out = false
x_flag = true
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0 , E_pot =  -20.7 , E_tot = -20.7
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
  for debugging purposes, here is the internal data representation:
  mass = 1.0
   pos = 1.0, 1.0, 1.0
   vel = 0.0, 0.0, 0.0
   acc = -0.705795165844984, -0.63811749631532, -0.604278661550489
  mass = 1.1
   pos = 1.0, 1.0, -1.0
   vel = 0.0, 0.0, 0.0
   acc = -0.725983913601737, -0.658306244072073, 0.556789739777578
  mass = 1.2
   pos = 1.0, -1.0, 1.0
   vel = 0.0, 0.0, 0.0
   acc = -0.74617266135849, 0.543139652769499, -0.644656157063995
  mass = 1.3
   pos = 1.0, -1.0, -1.0
   vel = 0.0, 0.0, 0.0
   acc = -0.766361409115244, 0.563328400526252, 0.597167235291084
  mass = 1.4
   pos = -1.0, 1.0, 1.0
   vel = 0.0, 0.0, 0.0
   acc = 0.515839478753342, -0.718872487342333, -0.685033652577501
  mass = 1.5
   pos = -1.0, 1.0, -1.0
   vel = 0.0, 0.0, 0.0
   acc = 0.536028226510095, -0.739061235099086, 0.637544730804591
  mass = 1.6
   pos = -1.0, -1.0, 1.0
   vel = 0.0, 0.0, 0.0
   acc = 0.556216974266848, 0.623894643796512, -0.725411148091007
  mass = 1.7
   pos = -1.0, -1.0, -1.0
   vel = 0.0, 0.0, 0.0
   acc = 0.576405722023601, 0.644083391553265, 0.677922226318097
at time t = 2, after 20 steps :
  E_kin = 1.58e+03 , E_pot =  -4.79 , E_tot = 1.57e+03
             E_tot - E_init = 1.6e+03
```

```
(E_tot - E_init) / E_init = -76.9
for debugging purposes, here is the internal data representation:
mass = 1.0
 pos = -3.02971661137991, -2.24193089826656, -2.25919153274194
 vel = -4.87425095159097, -3.66549478896126, -3.74246308641192
 acc = 0.0606083601111188, 0.0515883922084716, 0.0733897970440225
mass = 1.1
 pos = -4.31240799333488, -3.10616175910087, 3.14764781958622
 vel = -6.86534163590612, -5.02388641613739, 5.35062470064063
 acc = 0.0615581909052363, 0.0415529106299662, -0.0445228494603485
mass = 1.2
 pos = -1.62568960640959, 3.34494098204815, -1.6622442836446
 vel = -2.3870995554365, 5.56374518497225, -2.53222993754156
 acc = 0.00462833018233916, -0.139148363026155, 0.101906470797269
mass = 1.3
 pos = -1.50564040892264, 1.27774837383927, 0.881834075119233
 vel = -2.08881245118275, 2.16684233579503, 1.29915422585319
 acc = -0.0179375991256231, -0.0112123628543447, -0.0971526926290998
mass = 1.4
 pos = 2.32076951856892, -4.11004315117335, -4.08236760983636
 vel = 3.97924211487247, -6.25425810387294, -6.25833469237557
 acc = -0.0488286812931284, 0.0362709826406403, 0.0495677416334056
mass = 1.5
 pos = 2.94056101577017, -5.90875150100682, 5.68060097022845
 vel = 4.91294310127069, -8.99490408184316, 8.88351464915947
 acc = -0.0382215840668386, 0.0318234057191825, -0.0300100124207618
mass = 1.6
 pos = 6.9881569489367, 11.4122398477483, -16.2293505526342
 vel = 11.1558374351108, 17.787395714252, -24.9415505085132
 acc = -0.00580613947864572, -0.00923780971388437, 0.0119874756627306
mass = 1.7
 pos = -5.15261441447721, -2.6227584261269, 13.1802716803286
 vel = -7.5197594733857, -3.83125593317898, 20.3231933860916
 acc = 0.0143675680082912, 0.000307955710085254, -0.0376257105250154
   N = 8
time = 0.0
   N = 8
time = 2.0
```

8.4 Convergence

Alice: A veritable numerical explosion! Look, the total energy has changed from a negative value around -20 to an enormously large positive value. So much for energy conservation. Clearly we'll have to try a much smaller time step.

Bob: At least the particles have passed through the center, as you can see from the particles that reversed the signs of the values of the components of their position vectors – although some of the particles seem to have gone of in almost random directions, with great speed. Okay, let's make the time step a hundred times smaller. We did start off with a rather unrealistically large time step value, after all.

Alice: And let's cut down on the output for now, showing only the energy errors for a few runs. If we give a large value, say 10, for the snapshot output interval, no snapshot will appear during our run. Here we go again:

```
|gravity> ruby rknbody8d_driver.rb < cube1.in
dt = 0.001
dt_dia = 2
dt_out = 10
dt_end = 2
init_out = false
x_flag = false
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0 , E_pot =  -20.7 , E_tot = -20.7
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2, after 2000 steps :
  E_kin = 2.35e+03 , E_pot =  -7.15 , E_tot = 2.35e+03
             E_tot - E_init = 2.37e+03
  (E_tot - E_init) / E_init = -114
```

Bob: Still a disaster. Well, these particle energy errors have little meaning, of course, once they are larger than the original energy values. So we have no way of knowing how much smaller we'll have to make the time step. Let's just try a time step value that is a factor ten smaller.

```
|gravity> ruby rknbody8e_driver.rb < cube1.in
dt = 0.0001
dt_dia = 2
dt_out = 10
dt_end = 2
init_out = false
x_flag = false
method = rk4
at time t = 0, after 0 steps :
```

```
  E_kin = 0 , E_pot =  -20.7 , E_tot = -20.7
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2, after 20000 steps :
  E_kin = 65.3 , E_pot =  -86.1 , E_tot = -20.7
             E_tot - E_init = -0.00705
  (E_tot - E_init) / E_init = 0.00034
```

Alice: Much better already! It seems that we're finally converging. But I'd like to be sure. How about a time step that is smaller yet, by a factor two:

```
|gravity> ruby rknbody8f_driver.rb < cube1.in
dt = 5.0e-05
dt_dia = 2
dt_out = 10
dt_end = 2
init_out = false
x_flag = false
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0 , E_pot =  -20.7 , E_tot = -20.7
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2, after 40000 steps :
  E_kin = 76.7 , E_pot =  -97.5 , E_tot = -20.7
             E_tot - E_init = -0.000228
  (E_tot - E_init) / E_init = 1.1e-05
```

Bob: Convergence declared. Good! Normally, a fourth-order integrator should get a factor 16 more accurate when you half the time step, so this is very satisfactory.

Chapter 9

Softening

9.1 Close Encounters

Alice: I'm very glad to see that we can integrate eight bodies in a cold collapse system. This is quite a bit more demanding than integrating a handful of bodies in a virialized system. However, in both cases, sooner or later there will be close encounters between two or more of the particles. Our code will never be able to handle all of those close encounters. No matter how small a time step we give it, sooner or later there will be particles that approach each other closely enough to have a near miss that takes less time than the time step size. This will necessarily lead to large numerical errors.

Bob: This is of course why people have introduced variable time steps, as well as a whole order set of algorithmic tools to tame the unruly behavior of particles that get too close to the inverse square singularities of Newtonian gravity.

Alice: Soon we will introduce those extensions in our codes, but for now, there are more urgent things on our agenda. I guess we just have to live with it, and make sure the students realize that this first N-body tool is not to be trusted under all circumstances.

Bob: Hmm. I don't much like the idea of giving someone a tool that cannot be trusted. How about adding softening, as an option?

Alice: You mean to soften the potential, from an inverse square law to a form that remains finite in the center?

Bob: Indeed. We start from the singular Newtonian potential energy between two particles with positions $\mathbf{r}_i$ and $\mathbf{r}_j$ and masses M_i, M_j:

$$U(\mathbf{r}_i, \mathbf{r}_j) = G\frac{M_iM_j}{|\mathbf{r}_j - \mathbf{r}_i|} \tag{9.1}$$

The standard softening approach is to replace this by a regular variant, simply by adding the square of a small quantity ϵ:

$$U(\mathbf{r}_i, \mathbf{r}_j, \epsilon) = G\frac{M_i M_j}{(|\mathbf{r}_j - \mathbf{r}_i|^2 + \epsilon^2)^{1/2}} \tag{9.2}$$

When you differentiate this modified potential with respect to the position of a particle, you obtain a modified acceleration:

$$\frac{d^2}{dt^2}\mathbf{r}_i = G\sum_{\substack{j=1 \\ j\neq i}}^{N} M_j \frac{\mathbf{r}_j - \mathbf{r}_i}{(|\mathbf{r}_j - \mathbf{r}_i|^2 + \epsilon^2)^{3/2}} \tag{9.3}$$

And of course, in the limit that $\epsilon \to 0$, this last equation again returns to the Newtonian gravitational acceleration.

9.2 Fuzzy-Point Particles

Alice: Yes, this is what is often used in collisionless stellar dynamics, to suppress the effect of close encounters. I can't say I'm very happy with this softening approach, since it's not the real thing. It is purely a mathematical trick, to avoid numerical problems.

Bob: Well, you *can* give it a physical interpretation. Instead of using point particles, which are not very physical in the first place, each particle gets a more extended mass distribution. In fact, you can easily show that a softened potential corresponds to a mass distribution given by a polytrope of index five, better known as a Plummer mass distribution:

$$\rho(r) \;\propto\; \frac{1}{(r^2 + \epsilon^2)^{5/2}} \tag{9.4}$$

Alice: But look, your mass distribution stretches all the way to infinity! Even though most of the mass in concentrated in a small region, with a radius of order the softening length ϵ. You solution works, in the sense of avoiding singularities, and it gives a roughly reasonable answer, but it does come at the cost of smearing each particle all over space.

Bob: It would be quite easy to use a different mass distribution, corresponding a finite support. This is what people so who work with SPH particles, for example. However, for or current purpose, the main thing is to provide a tool that works, and we can worry later about aesthetic details.

Alice: Okay. Even though I can't say I'm very happy with it, I see your point, and it is certainly safer to give the students a tool that is guaranteed to give finite answer.

Bob: It should be easy to add softening to our code. Time to create another version for our N-body code! So we will call this new file `rknbody9.rb`. Well, this will take me a while.

Alice: Okay, I'm way behind in reading the astro-ph abstracts. This will give me a chance to catch up. I'll come back when I've gone through them.

9.3 A New Driver

Bob: Here it is, the new version of our N-body code, now with softening build in. It was quite straightforward to make the changes. First of all, here is the new driver:

```
require "rknbody9.rb"

include Math

eps = 0
dt = 0.001           # time step
dt_dia = 2.1088      # diagnostics printing interval
dt_out = 2.1088      # output interval
dt_end = 2.1088      # duration of the integration
init_out = false     # initial output requested ?
x_flag = false       # extra diagnostics requested ?
##method = "forward"  # integration method
##method = "leapfrog" # integration method
##method = "rk2"      # integration method
method = "rk4"       # integration method

STDERR.print "eps = ", eps, "\n",
      "dt = ", dt, "\n",
      "dt_dia = ", dt_dia, "\n",
      "dt_out = ", dt_out, "\n",
      "dt_end = ", dt_end, "\n",
      "init_out = ", init_out, "\n",
      "x_flag = ", x_flag, "\n",
      "method = ", method, "\n"

nb = Nbody.new
nb.simple_read
nb.evolve(method, eps, dt, dt_dia, dt_out, dt_end, init_out, x_flag)
```

As you can see, minimal differences, contained in three lines. The method `evolve` has an extra parameter, `eps`, the softening length. The default value is zero, which means no softening at all. The third new line is where the value of `eps` is echoed on the standard error stream.

Alice: So now `evolve` has eight parameters. At some point we may want to think about grouping them together, perhaps creating a class for them, since there is clear substructure: two flags controlling the amount of output, three variables giving intervals between output times, and three other variables.

Bob: But not now.

Alice: Not now, no. Can you show me the code itself?

9.4 A Code with Softening

Bob: Here it is. Almost all changes speak for themselves.

```
require "vector.rb"

class Body

  attr_accessor :mass, :pos, :vel

  def initialize(mass = 0, pos = Vector[0,0,0], vel = Vector[0,0,0])
    @mass, @pos, @vel = mass, pos, vel
  end

  def calc(softening_parameter, body_array, time_step, s)
    ba  = body_array
    dt = time_step
    eps = softening_parameter
    eval(s)
  end

  def acc(body_array, eps)
    a = @pos*0                              # null vector of the correct length
    body_array.each do |b|
      unless b == self
        r = b.pos - @pos
        r2 = r*r + eps*eps
        r3 = r2*sqrt(r2)
        a += r*(b.mass/r3)
      end
    end
    a
  end
```

```
  def ekin                         # kinetic energy
    0.5*@mass*(@vel*@vel)
  end

  def epot(body_array, eps)        # potential energy
    p = 0
    body_array.each do |b|
      unless b == self
        r = b.pos - @pos
        p += -@mass*b.mass/sqrt(r*r + eps*eps)
      end
    end
    p
  end

  def to_s
    "  mass = " + @mass.to_s + "\n" +
    "   pos = " + @pos.join(", ") + "\n" +
    "   vel = " + @vel.join(", ") + "\n"
  end

  def pp                           # pretty print
    print to_s
  end

  def ppx(body_array, eps)         # pretty print, with extra information (acc)
    STDERR.print to_s + "   acc = " + acc(body_array, eps).join(", ") + "\n"
  end

  def simple_print
    printf("%24.16e\n", @mass)
    @pos.each{|x| printf("%24.16e", x)}; print "\n"
    @vel.each{|x| printf("%24.16e", x)}; print "\n"
  end

  def simple_read
    @mass = gets.to_f
    @pos = gets.split.map{|x| x.to_f}.to_v
    @vel = gets.split.map{|x| x.to_f}.to_v
  end

end

class Nbody

  attr_accessor :time, :body
```

```
def initialize
  @body = []
end

def evolve(integration_method, eps, dt, dt_dia, dt_out, dt_end,
           init_out, x_flag)
  @dt = dt
  @eps = eps
  nsteps = 0
  e_init
  write_diagnostics(nsteps, x_flag)
  t_dia = dt_dia - 0.5*dt
  t_out = dt_out - 0.5*dt
  t_end = dt_end - 0.5*dt

  simple_print if init_out

  while @time < t_end
    send(integration_method)
    @time += dt
    nsteps += 1
    if @time >= t_dia
      write_diagnostics(nsteps, x_flag)
      t_dia += dt_dia
    end
    if @time >= t_out
      simple_print
      t_out += dt_out
    end
  end
end

def calc(s)
  @body.each{|b| b.calc(@eps, @body, @dt, s)}
end

def forward
  calc(" @old_acc = acc(ba,eps) ")
  calc(" @pos += @vel*dt ")
  calc(" @vel += @old_acc*dt ")
end

def leapfrog
  calc(" @vel += acc(ba,eps)*0.5*dt ")
  calc(" @pos += @vel*dt ")
```

```
    calc(" @vel += acc(ba,eps)*0.5*dt ")
  end

  def rk2
    calc(" @old_pos = @pos ")
    calc(" @half_vel = @vel + acc(ba,eps)*0.5*dt ")
    calc(" @pos += @vel*0.5*dt ")
    calc(" @vel += acc(ba,eps)*dt ")
    calc(" @pos = @old_pos + @half_vel*dt ")
  end

  def rk4
    calc(" @old_pos = @pos ")
    calc(" @a0 = acc(ba,eps) ")
    calc(" @pos = @old_pos + @vel*0.5*dt + @a0*0.125*dt*dt ")
    calc(" @a1 = acc(ba,eps) ")
    calc(" @pos = @old_pos + @vel*dt + @a1*0.5*dt*dt ")
    calc(" @a2 = acc(ba,eps) ")
    calc(" @pos = @old_pos + @vel*dt + (@a0+@a1*2)*(1/6.0)*dt*dt ")
    calc(" @vel += (@a0+@a1*4+@a2)*(1/6.0)*dt ")
  end

  def ekin                        # kinetic energy
    e = 0
    @body.each{|b| e += b.ekin}
    e
  end

  def epot                        # potential energy
    e = 0
    @body.each{|b| e += b.epot(@body, @eps)}
    e/2                           # pairwise potentials were counted twice
  end

  def e_init                      # initial total energy
    @e0 = ekin + epot
  end

  def write_diagnostics(nsteps, x_flag)
    etot = ekin + epot
    STDERR.print <<END
at time t = #{sprintf("%g", time)}, after #{nsteps} steps :
  E_kin = #{sprintf("%.3g", ekin)} ,\
 E_pot =  #{sprintf("%.3g", epot)} ,\
 E_tot = #{sprintf("%.3g", etot)}
             E_tot - E_init = #{sprintf("%.3g", etot - @e0)}
```

```
  (E_tot - E_init) / E_init = #{sprintf("%.3g", (etot - @e0)/@e0 )}
END
    if x_flag
      STDERR.print "  for debugging purposes, here is the internal data ",
                   "representation:\n"
      ppx
    end
  end

  def pp                          # pretty print
    print "     N = ", @body.size, "\n"
    print "  time = ", @time, "\n"
    @body.each{|b| b.pp}
  end

  def ppx                         # pretty print, with extra information (acc)
    print "     N = ", @body.size, "\n"
    print "  time = ", @time, "\n"
    @body.each{|b| b.ppx(@body, @eps)}
  end

  def simple_print
    print @body.size, "\n"
    printf("%24.16e\n", @time)
    @body.each{|b| b.simple_print}
  end

  def simple_read
    n = gets.to_i
    @time = gets.to_f
    for i in 0...n
      @body[i] = Body.new
      @body[i].simple_read
    end
  end

end
```

9.5 Details

Alice: Even though the changes may speak for themselves, I have some questions. First of all, the value of `eps` has to be passed on from the driver, where it is defined, through `evolve` into `Nbody` and then down to the methods within `Body` that do all the hard work.

Bob: First of all, I gave the Nbody class an extra instance variable, @eps, which stores the value of the softening. As soon as evolve is executed, within the Nbody class, the first thing it does is assign the proper value to @eps, as well as to @dt, as was done already in our previous version:

```
@dt = dt
@eps = eps
```

Alice: I see. In that way, you don't have to give an extra argument to the integration methods, for example: they can just pick up the value of the softening length from @eps, to which they have automatic access, as Nbody class methods. But of course they *do* have to pass that value down to the particles, which are realized as instances of the Body class, since otherwise the particles would not know what softening to use.

Bob: The one thing I didn't like very much is that the lines in the integration methods have become somewhat longer. Forward Euler, for example, has grown now from:

```
def forward
  calc(" @old_acc = acc(ba) ")
  calc(" @pos += @vel*dt ")
  calc(" @vel += @old_acc*dt ")
end
```

to:

```
def forward
  calc(" @old_acc = acc(ba,eps) ")
  calc(" @pos += @vel*dt ")
  calc(" @vel += @old_acc*dt ")
end
```

I'm not too happy with the fact that acc now has to get a second argument. But that's the way it is.

If I *really* would want to make the lines shorter, I could use shorter variables than ba and eps, for example a and e. But let us not spend more time on such niceties, which give us diminishing return in clarity at the cost of making the code more complex and hence less clear.

Alice: I fully agree. Instead, let's see how your new code performance in our cold collapse experiment.

Chapter 10

Cold Collapse with Softening

10.1 Check

Alice: It's probably a good idea to try our standard check with the figure-8 three-body system, just to make sure that with zero softening we get the same results as before.

Bob: Yes, I agree. Here we go:

```
|gravity> ruby rknbody9a_driver.rb < figure8.in
eps = 0
dt = 0.001
dt_dia = 2.1088
dt_out = 2.1088
dt_end = 2.1088
init_out = false
x_flag = false
method = rk4
at time t = 0, after 0 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2.109, after 2109 steps :
  E_kin = 1.21 , E_pot =  -2.5 , E_tot = -1.29
             E_tot - E_init = -2e-15
  (E_tot - E_init) / E_init = 1.55e-15
3
  2.1089999999998787e+00
```

```
  1.0000000000000000e+00
 -1.6047303546488470e-04 -1.9320664965417420e-04
 -9.3227640249930266e-01 -8.6473492670753516e-01
  1.0000000000000000e+00
  9.7020367429337440e-01 -2.4296620300772800e-01
  4.6595057278750124e-01  4.3244644507801255e-01
  1.0000000000000000e+00
 -9.7004320125790211e-01  2.4315940965738195e-01
  4.6632582971180025e-01  4.3228848162952316e-01
```

10.2 Large Softening

Alice: Good. Now let's try the same cold collapse as before, but with a softening length of, say, 0.1. At the beginning of their free fall, the particles will almost feel the same forces as they did before. Let us compare it with the run that had a time step of 0.001. For that case we had a horrible lack of energy conservation. Your softened code should do a lot better.

Bob: Okay, let us see:

```
|gravity> ruby rknbody9b_driver.rb < cube1.in
eps = 0.1
dt = 0.001
dt_dia = 2
dt_out = 2
dt_end = 2
init_out = false
x_flag = false
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0 , E_pot =  -20.7 , E_tot = -20.7
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2, after 2000 steps :
  E_kin = 48.1 , E_pot =  -68.8 , E_tot = -20.7
             E_tot - E_init = -1e-05
  (E_tot - E_init) / E_init = 4.84e-07
8
  1.9999999999998905e+00
  1.0000000000000000e+00
 -2.3357068379703563e+00 -1.5730462500735882e+00 -1.8373515169655967e+00
 -3.2503099351457405e+00 -2.2769160097020231e+00 -2.7623089808977994e+00
  1.1000000000000001e+00
```

```
-1.4462439521294790e+00 -7.9274043868280397e-01  1.9462651835363280e+00
-1.5388550025735492e+00 -8.5033761684692111e-01  2.7987214535075422e+00
 1.2000000000000000e+00
-8.4361749490035387e-01  1.8870481468897002e+00 -7.5123041488377962e-01
-5.2602892044713823e-01  2.5585640337825617e+00 -6.5671580935516860e-01
 1.3000000000000000e+00
 1.6304345239074072e-01  5.8421169904952219e-01  4.1153090918721674e-01
 1.3842631574595166e+00 -5.2283176060070413e-01 -5.2052114949956041e-01
 1.3999999999999999e+00
 3.0263433758971969e-01 -1.2126952329091388e-01  5.4757741932432893e-02
-2.2026773446067907e+00 -1.0870392845510134e-01 -2.4523203993014261e-01
 1.5000000000000000e+00
 5.7619765613623819e-01 -1.4294804330220603e-01 -9.1132984353013688e-02
-1.0609857121621187e+00 -1.6441584074724240e+00 -9.9518893248749163e-01
 1.6000000000000001e+00
 5.5584843594067390e-01 -3.6360342031389209e-01 -2.0891409427520066e-01
 2.3016236123480418e+00  1.1615617212579910e+00 -5.4794699774267885e-01
 1.7000000000000000e+00
 5.5859860985306420e-01 -2.4288669549678879e-01  3.3674893916023442e-02
 2.8043365345598055e+00  9.3036246676969148e-01  2.2713384109628652e+00
```

Alice: Great! Very well behaved. And indeed most particles have just passed through the center, as is clear from their position components, and are continuing to move on, as their velocity components indicate.

But wait a minute, the energy `E_tot` at time `t = 0` is the same as before. How can that be? When we change the potential, there should at least be some change in the value of the initial total energy.

Bob: Ah, but the particles are all separated by at least 2 length units from each other. Since the softening always comes in through an expression containing $r^2+\epsilon^2$, we have to check the difference between 2^2 and $2^2+(0.1)^2$. The latter is only a quarter of a percent larger than the former. And for most particle pairs the difference is much smaller still, so the total difference is likely to be more like a tenth of a percent, too small to show up within the accuracy with which we print out the energy.

10.3 Even larger softening

Alice: just to make sure, let us take a softening length of 0.3. According to your analysis, that would show a difference in the initial total energy, right?

Bob: I would think so. Okay, let's try:

```
|gravity> ruby rknbody9c_driver.rb < cube1.in
```

```
eps = 0.3
dt = 0.001
dt_dia = 2
dt_out = 2
dt_end = 2
init_out = false
x_flag = false
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0 , E_pot =  -20.6 , E_tot = -20.6
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2, after 2000 steps :
  E_kin = 19.9 , E_pot =  -40.5 , E_tot = -20.6
             E_tot - E_init = -6.68e-09
  (E_tot - E_init) / E_init = 3.25e-10
8
  1.9999999999998905e+00
  1.0000000000000000e+00
 -1.3435332524361765e+00 -1.3215614066272325e+00 -1.3349638266570156e+00
 -1.6912633178958605e+00 -1.8593451296223482e+00 -1.9880447778358679e+00
  1.1000000000000001e+00
 -1.1428786019621284e+00 -1.1455131430342225e+00  1.2256328116240207e+00
 -1.1933788723000605e+00 -1.3945830515013942e+00  1.8622141170854261e+00
  1.2000000000000000e+00
 -9.2565210028918554e-01  1.0727125108268556e+00 -9.6980727607605521e-01
 -6.6075234951751749e-01  1.5546315810261460e+00 -1.0191929255321155e+00
  1.3000000000000000e+00
 -6.7586116933868967e-01  8.5050524262812943e-01  8.0855082812733969e-01
 -5.1962467791243423e-02  1.0021841594452070e+00  7.8942187688433141e-01
  1.3999999999999999e+00
  7.4446049432842354e-01 -4.9008095056780804e-01 -5.0941448582820292e-01
  8.6641095431182236e-01  4.6983803735978924e-01  3.8723066728293609e-01
  1.5000000000000000e+00
  5.2586784274845078e-01 -1.9076507421867578e-01  3.0256970789252191e-01
  3.7062049124660296e-01  1.3593990088399948e+00 -9.4199154209711755e-01
  1.6000000000000001e+00
  3.4351961512732321e-01  1.1579367694820625e-01  8.9914604762325170e-02
  2.1678167378492888e-01 -1.3136771019591300e+00  1.4926999954472413e+00
  1.7000000000000000e+00
  3.5848465570332533e-01  1.0335593797758784e-01 -1.0889250629570488e-01
  1.0286337376548356e+00 -2.1764660039138783e-01 -8.1239017296188831e-01
```

Alice: Good! I'm glad to see that.

Bob: Yes, it never hurts to check.

Alice: And it hurts a lot if you don't check, and run into mysterious problems later.

10.4 Small Softening

Bob: Let's see how far we can push it. How about a softening length of 0.01? And let me suppress the snapshot output for now:

```
|gravity> ruby rknbody9d_driver.rb < cube1.in
eps = 0.01
dt = 0.001
dt_dia = 2
dt_out = 10
dt_end = 2
init_out = false
x_flag = false
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0 , E_pot =  -20.7 , E_tot = -20.7
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2, after 2000 steps :
  E_kin = 227 , E_pot =  -804 , E_tot = -577
             E_tot - E_init = -556
  (E_tot - E_init) / E_init = 26.8
```

Alice: Not too surprising. With velocities of order unity, and a softening length that is only ten times larger than the time step, a typical particle will step through the core of the potential of another particle in only a few steps.

Bob: And when doing so, the particle will be sped up already to velocities typical well above unity, leaving even fewer integration steps during which the attraction between the particles changes dramatically. In fact, when they approach each other to a distance of order 0.01, there speed will be at least 10 in our units, and probably larger than that. Two particles may pass each other through their softening radius in even less than one time step.

So yes, it would have been worrisome if the errors would *not* have been large. Let me use a ten times smaller time step:

```
|gravity> ruby rknbody9e_driver.rb < cube1.in
eps = 0.01
dt = 0.0001
dt_dia = 2
```

```
dt_out = 10
dt_end = 2
init_out = false
x_flag = false
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0 , E_pot =  -20.7 , E_tot = -20.7
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2, after 20000 steps :
  E_kin = 87.4 , E_pot =  -108 , E_tot = -20.7
             E_tot - E_init = -0.00458
  (E_tot - E_init) / E_init = 0.000221
```

Alice: That's more like it. Actually, not so different from what we found earlier, without softening.

Bob: I guess this means that our eight particles did not come much closer to each other than distances of order 0.01.

Alice: Which is reasonable. In three dimensions there are two independent directions in which two approaching particles can miss each other, and you have to aim carefully to come really close.

Bob: Let me make the time step half as small again, just to check whether the error in the energy conservation drops by at least a factor of sixteen:

```
|gravity> ruby rknbody9f_driver.rb < cube1.in
eps = 0.01
dt = 5.0e-05
dt_dia = 2
dt_out = 10
dt_end = 2
init_out = false
x_flag = false
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0 , E_pot =  -20.7 , E_tot = -20.7
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2, after 40000 steps :
  E_kin = 101 , E_pot =  -122 , E_tot = -20.7
             E_tot - E_init = -0.000141
  (E_tot - E_init) / E_init = 6.81e-06
```

Alice: I think we can declare victory.

10.5 Central Collapse

Bob: Ah, let's do something fun, something we never could have done without softening. Let us give all particles equal masses, so that when they drop from the corners of the cube, they all will reach each other at the center. Even so, softening should keep them from misbehaving.

Alice: As you like!

Bob: So these are the new initial conditions:

```
8
0
1
1 1 1
0 0 0
1
1 1 -1
0 0 0
1
1 -1 1
0 0 0
1
1 -1 -1
0 0 0
1
-1 1 1
0 0 0
1
-1 1 -1
0 0 0
1
-1 -1 1
0 0 0
1
-1 -1 -1
0 0 0
```

And this is the result:

```
|gravity> ruby rknbody9b_driver.rb < cube2.in
eps = 0.1
dt = 0.001
dt_dia = 2
dt_out = 2
```

```
dt_end = 2
init_out = false
x_flag = false
method = rk4
at time t = 0, after 0 steps :
  E_kin = 0 , E_pot =  -11.4 , E_tot = -11.4
             E_tot - E_init = 0
  (E_tot - E_init) / E_init = -0
at time t = 2, after 2000 steps :
  E_kin = 8.54 , E_pot =  -19.9 , E_tot = -11.4
             E_tot - E_init = -4.77e-05
  (E_tot - E_init) / E_init = 4.19e-06
8
  1.9999999999998905e+00
  1.0000000000000000e+00
 -5.7041409218602102e-01 -5.7041409218602102e-01 -5.7041409218602102e-01
 -8.4345690649961835e-01 -8.4345690649961835e-01 -8.4345690649961835e-01
  1.0000000000000000e+00
 -5.7041409218602102e-01 -5.7041409218602102e-01  5.7041409218602102e-01
 -8.4345690649961835e-01 -8.4345690649961835e-01  8.4345690649961835e-01
  1.0000000000000000e+00
 -5.7041409218602090e-01  5.7041409218602102e-01 -5.7041409218602102e-01
 -8.4345690649961791e-01  8.4345690649961835e-01 -8.4345690649961835e-01
  1.0000000000000000e+00
 -5.7041409218602102e-01  5.7041409218602102e-01  5.7041409218602102e-01
 -8.4345690649961835e-01  8.4345690649961835e-01  8.4345690649961835e-01
  1.0000000000000000e+00
  5.7041409218602102e-01 -5.7041409218602090e-01 -5.7041409218602090e-01
  8.4345690649961835e-01 -8.4345690649961791e-01 -8.4345690649961791e-01
  1.0000000000000000e+00
  5.7041409218602102e-01 -5.7041409218602102e-01  5.7041409218602090e-01
  8.4345690649961835e-01 -8.4345690649961835e-01  8.4345690649961791e-01
  1.0000000000000000e+00
  5.7041409218602090e-01  5.7041409218602090e-01 -5.7041409218602102e-01
  8.4345690649961791e-01  8.4345690649961791e-01 -8.4345690649961835e-01
  1.0000000000000000e+00
  5.7041409218602102e-01  5.7041409218602102e-01  5.7041409218602102e-01
  8.4345690649961835e-01  8.4345690649961835e-01  8.4345690649961835e-01
```

Alice: Well behaved indeed. Glad to see it all works!

Chapter 11

Literature References

[to be provided]